Лист, камен, ножички игра

Дигитално процесирање на слика

Кристијан Кузмановски 163137

Играта „лист камен ножички“ е доста интуитивна и лесна игра за луѓето, но не толку за компјутерите. Во овој труд ќе ви го опишам мојот пристап кон имплементирање на играта „лист камен ножички“ во компјутерски код.

При креирање на играта се соочив со следниве проблеми:

1. Оделување на позадината од раката
2. Одредување на формата на раката (дали е во форма на лист или камен …)
3. Како компјутерот да направи свој потег

Оделување на позадината од раката

При пребарување низ интернет за најдобриот алгоритам за мојата потреба наидов на многу алгоритми секои со свои предности и препреки. Наидов на алгоритми кои се потпираат на вештачка интелигенција и класификација за да ја откријат раката и да ја одвојат од позадината. Но за употреба на овие алгоритми беше потребно најпрвин да се истренираат алгоритмите и класификаторите. Други алгоритми беа многу захтевни на ресурсите. Идеалниот алгоритам за мојата употреба е оној кој не бара многу ресурси се адаптира на различни околини и е брз за иницјализација. Открив неколку алгоритми кои ги исполнуваат овие критериуми:

1. Canny Edge detection

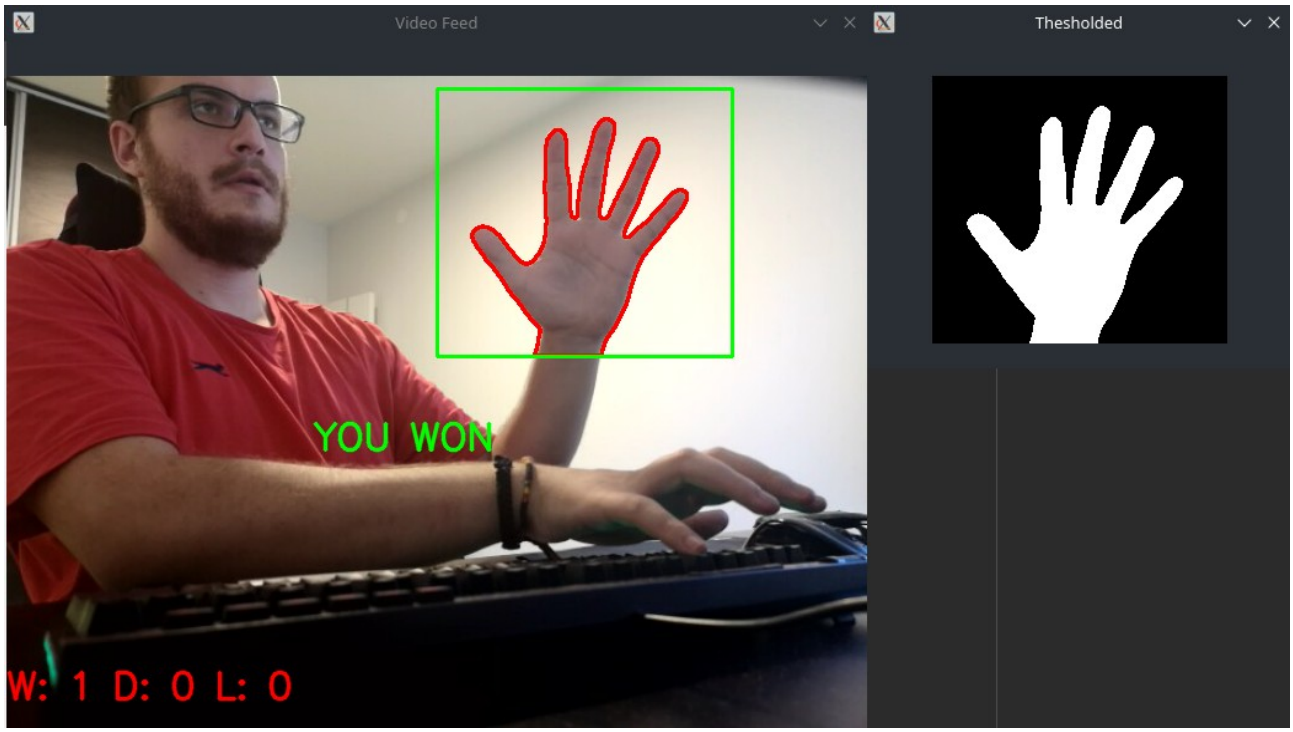
Овој алгоритам е еден од првите алгоритми на кои ќе ни текне за имплементација во вакава игра. Логиката позади ваквото размислување е бидејќи бојата на раката е доста уникатна во споредба со работите кои би можеле да ги најдеме во позадина ова би ни овозможило со помош на тој контраст да ја детектираме раката доста лесно.

Но како што можете да видите од сликата, алгоритамот доста подлежи на шум како и на лошо осветлување. Овие препреки го прават несоодветен за имплементирање во нашата игра.

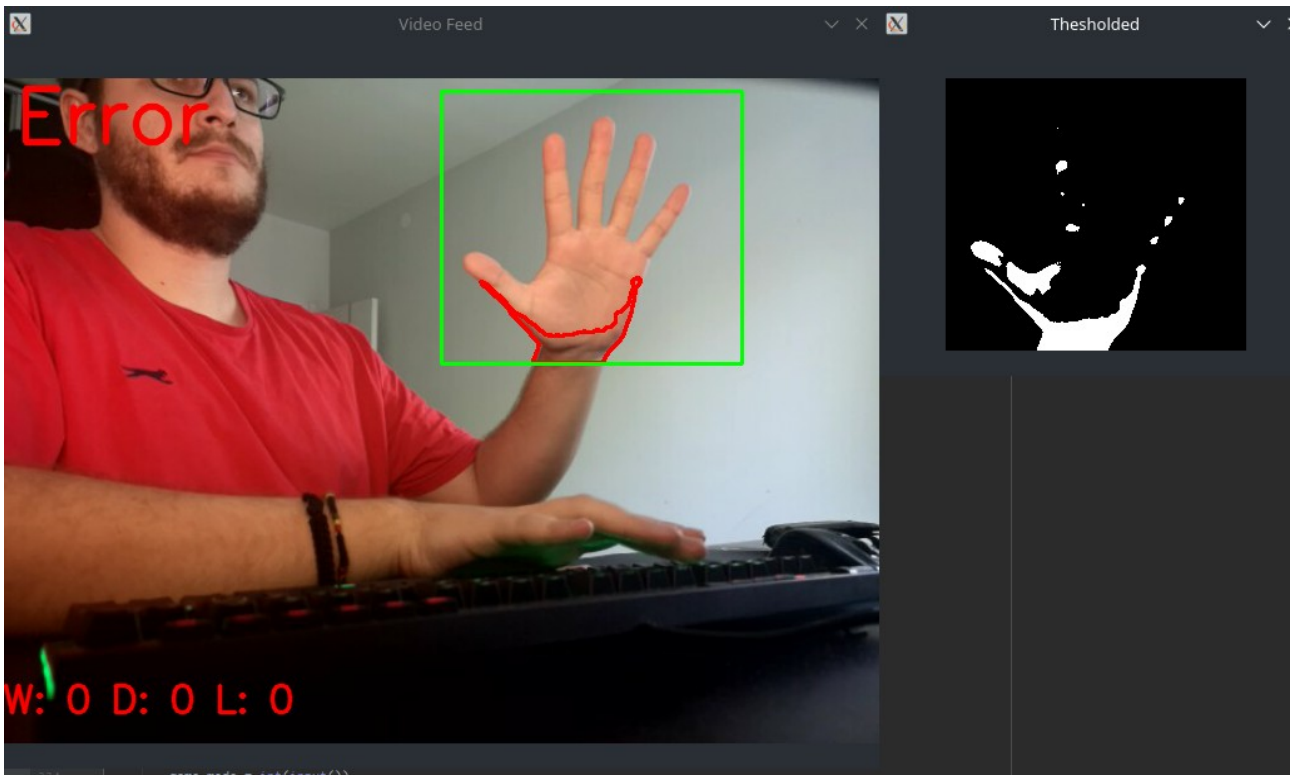
2. Background Subtraction

Овој алгоритам функционра на тој начин што имаме период на калибрација каде што се зимаат и процесираат слики од одредена област. На секоја слика се пресметува просекот помеѓу неа и предходната. Овие проседи се аграгираат. Теоретски овој алгоритам би можел да работи перфектно во секоја околина без разлика од осветлувањето. Но како што ќе видиме и овој алгоритам има недостатоци.

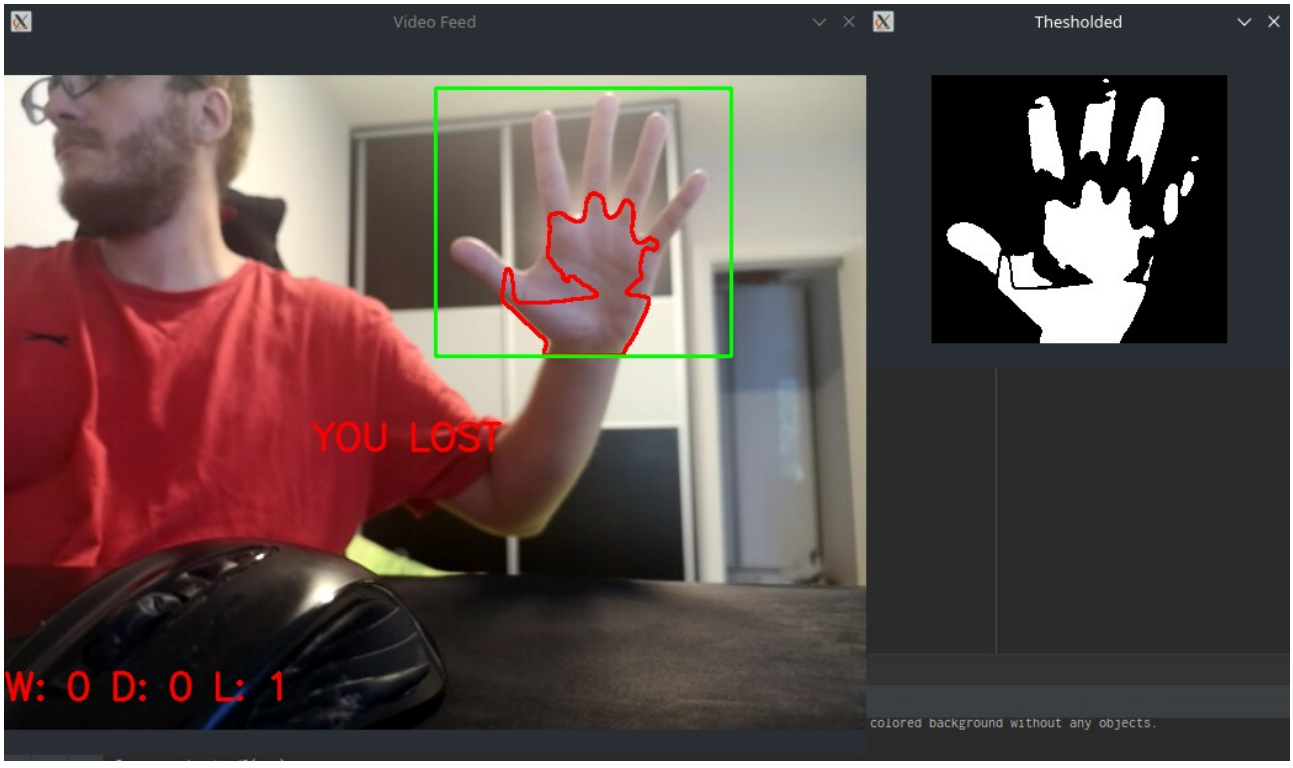
При правилна калибрација, соодветна околина и доволно осветлување алгоритамот совршено ја одвојува раката од позадината.



Но исто како и минатиот алгоритам подлежи на проблеми со лошо осветлување.



Иако алгоритамот не е многу осетлив на шум тој не работи совршено кога во позадината има темни објекти.



Алгоритамот не работи ако во регионот на интерес има повеќе подвижни објекти. За најдобри резултати потребна е добро осветлена статичка позадина (без темни објекти во позадина) каде единствен објект што се движи е раката.

Background Subtraction алгоритамот има доста недостатоци и не се совршен, но е доволно отпорен на шум, едноставен за имплементирање и лесен на ресурсите што го прави идеален за мојата игра.

Импелментирање на Background Subtraction

Најпрвио ги импортираме потребните библиотеки и декларираме глобална променлива find_avg_bg во која ќе се содржи агрегираната просечна вредност на позадината.

```
# Average background var
avg_bg = None
# Difficulty mode
```

Потоа ја дефинираме функцијата која ќе ја повикуваме при процесот на калибрирање.

```
# Find the average background
def find_avg_bg(image, weight):
    global avg_bg
    # initialize the background
    if avg_bg is None:
        avg_bg = image.copy().astype("float")
        return

    # compute weighted average, accumulate it and update the background
    cv2.accumulateWeighted(image, avg_bg, weight)
```

Оваа функција прима два аргументи

image – сегашниот фрејм

weight – тежината или вредноста која означува колку брзо да се забораваат предходните фрејмови

Функцијата проверува дали глобалната променлива е иницијализирана ако не е тогаш ја иницијализира и со помош на accumulateWeighted го пресметува просегот помеѓу фрејмовите.

Внатре во main функцијата ги дефинираме:

weight – тежината за во find_avg_bg функцијата

camera – референца до камерата

top,right,bottom,left – дефинирање на крајните точки на регионот од интерес

num_frames – бројач на фрејмови кој ќе помофне при калибрирање на алгоритамот

```
# initialize accumulated weight
weight = 0.5

# get the reference to the webcam
camera = cv2.VideoCapture(0)

# region of interest (ROI) coordinates
top, right, bottom, left = 10, 350, 225, 590

# initialize num of frames
num_frames = 0
```

Потоа креираме безкрајен loop со помош на while(True) кој претатавува game loop во кој главната логика на играта се извршува.

```
# keep looping, until interrupted
while (True):
    # get the current frame
    (grabbed, frame) = camera.read()

    # resize the frame
    frame = imutils.resize(frame, width=700)

    # flip the frame so that it is not the mirror view
    frame = cv2.flip(frame, 1)

    # clone the frame
    clone = frame.copy()

    # get the ROI
    roi = frame[top:bottom, right:left]

    # convert the roi to grayscale and blur it
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    gray = cv2.GaussianBlur(gray, (7, 7), 0)

    # to get the background, keep looking till a threshold is reached
    # so that our weighted average model gets calibrated
    if num_frames < 30:
        find_avg_bg(gray, weight)
        if num_frames == 1 and debug:
            print("Please wait while the program is calibrating...")
        elif num_frames == 29 and debug:
            print("Calibration successfull!")
```

На почетокот од game loop-от ние го земаме фрејмот од камерата ширината ја ставаме да биде 700 пиксели го превртуваме фрејмот и правиме копија од него. Потоа од фрејмот го оделуваме регионот од интерес тоа е делот означен со зелен квадрат во кој треба да ја покажеме раката. Одкога ќе го одвоиме регионот од интерес од фрејмот го конвертитаме во gray scale и го поминуваме со gaussian blur за да го „измазниме“. Доколку програмата е во

текот на калибрација (во мојата имплементација на играта периодот на калибрација е првите 30 фрејмови) ја повикуваме функцијата find_avg_bg, но ако програмата е веќе калибрирана тогаш пробуваме да ја детектираме и сегментираме раката.

Сегментацијата на раката ја вршиме со помош на функцијата find_hand.

```
# Find the hand
def find_hand(image, threshold=25):
    global avg_bg
    # find the absolute difference between background and current frame
    diff = cv2.absdiff(avg_bg.astype("uint8"), image)

    # threshold the diff image so that we get the foreground
    thresholded = cv2.threshold(diff, threshold, 255, cv2.THRESH_BINARY)[1]

    # get the contours in the thresholded image
    cnts = cv2.findContours(thresholded.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[0]

    # return None, if no contours detected
    if len(cnts) == 0:
        return
    else:
        # based on contour area, get the maximum contour which is the hand
        segmented = max(cnts, key=cv2.contourArea)
        return (thresholded, segmented)
```

Сегментирањето на раката од позадината е прилично едноставен процес. Најпрво ја наоѓаме абсолутната разлика помеѓу сегашниот фрејм (кој го задаваме како аргумент) и просечната позадина. Потоа ја бинарно treshold-нуваме разликата и потоа ги наоѓаме контурите на резултатот од treshold-нувањето.

Доколку има пронајдени контури ја земаме најголемата.

Одредување на формата на раката

Одредувањето на формата на раката може да се изврши само по сегментацијата на раката од позадината. Квалитетот на одредувањето на формата е директно поврзан со квалитетот на сегментацијата. Затоа големиот дел од мојот фокус беше во наоѓање на правилниот алгоритам на сегментација.

За одредување на формата ги бројам прстите на раката. Ако формата е камен тогаш алгоритамот не би требало да детектира ниеден прст, ако е лист токаш треба да детектира пет прста доколку пак формата е ножички тогаш алгоритамот ќе детектира два прста.

За пребројување на прстите ја користам функцијата count_fingers која како аргументи прима

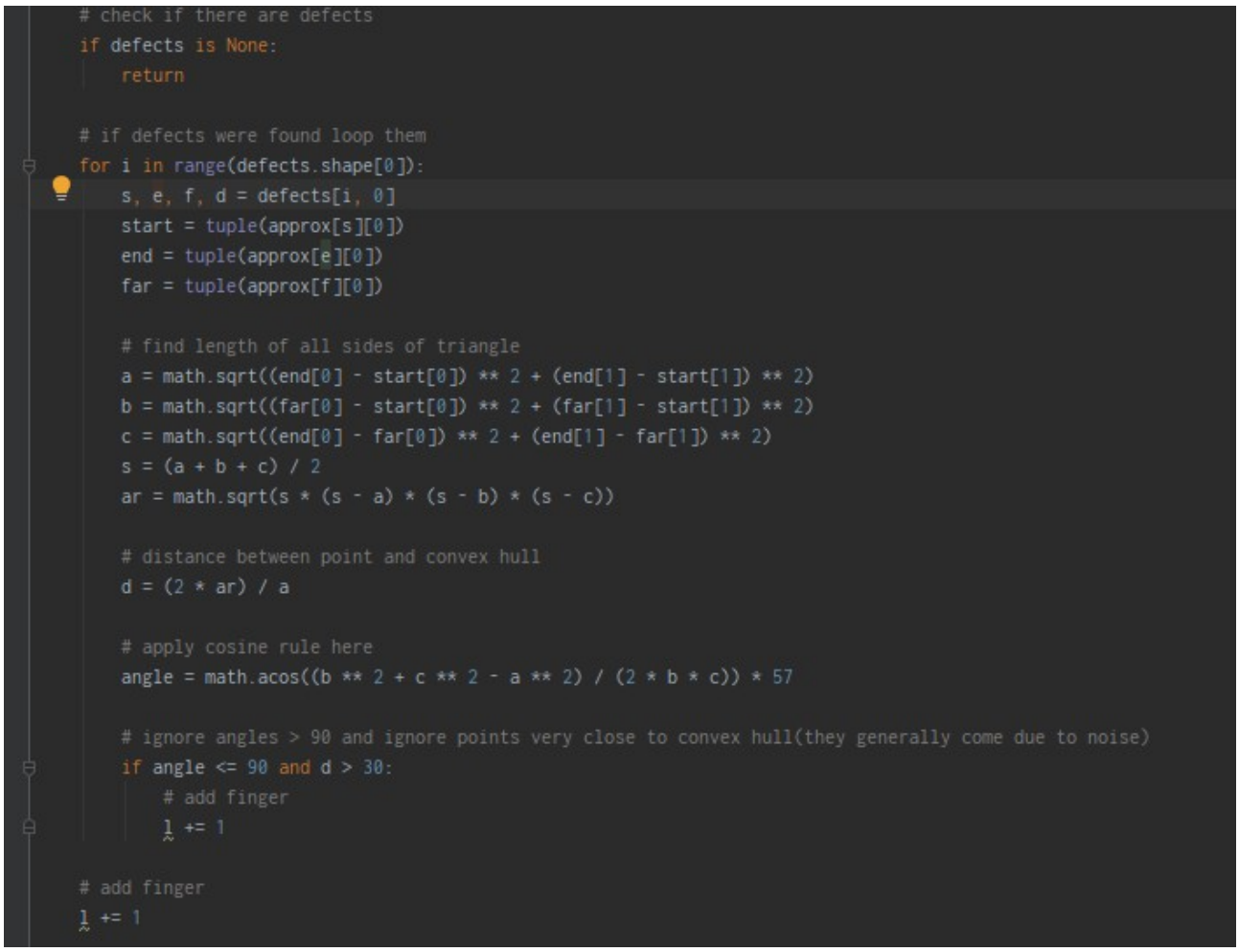
segmented - сегментираната слика од раката

frame – фрејм од камерата

```
def count_fingers(segmented, frame):
    global rock_count,scissors_count,paper_count

    # smoothen the contour a little
    epsilon = 0.0005 * cv2.arcLength(segmented, True)
    approx = cv2.approxPolyDP(segmented, epsilon, True)

    # make convex hull around hand
    hull = cv2.convexHull(segmented)

    # define area of hull and area of hand
    areahull = cv2.contourArea(hull)
    areacnt = cv2.contourArea(segmented)

    # check the we are not deviding by 0
    if areacnt != 0:
        # find the percentage of area not covered by hand in convex hull
        arearatio = ((areahull - areacnt) / areacnt) * 100

    # make convex hull around the smoothend contour of the hand
    hull = cv2.convexHull(approx, returnPoints=False)

    # prevent "The convex hull indices are not monotonous" error
    try:
        # find the defects in convex hull with respect to hand
        defects = cv2.convexityDefects(approx, hull)
    except:
        if debug:
            print("ERROR")
        return
    # var that will hold the number of fingers
    l=0

    # check if there are defects
    if defects is None:
        return
```

Функцијата најпрво ја измазнува сегментираната рака со помош на функцијата approxPolyDP.

Потоа ја наоѓаме конвексната обвивка на раката и ги одредуваме областа на обвивката и областа на сегментираната рака. Со овие две области го наоѓаме процентот на обвивката кој не е зафатен од сегментираната рака.Потоа креираме уште една конвексна обвивка на измазнетета рака. Со помош на оваа обвивка ние ги бараме конвексните дефекти.



```
# check if there are defects
if defects is None:
    return

# if defects were found loop them
for i in range(defects.shape[0]):
    s, e, f, d = defects[i, 0]
    start = tuple(approx[s][0])
    end = tuple(approx[e][0])
    far = tuple(approx[f][0])

    # find length of all sides of triangle
    a = math.sqrt((end[0] - start[0]) ** 2 + (end[1] - start[1]) ** 2)
    b = math.sqrt((far[0] - start[0]) ** 2 + (far[1] - start[1]) ** 2)
    c = math.sqrt((end[0] - far[0]) ** 2 + (end[1] - far[1]) ** 2)
    s = (a + b + c) / 2
    ar = math.sqrt(s * (s - a) * (s - b) * (s - c))

    # distance between point and convex hull
    d = (2 * ar) / a

    # apply cosine rule here
    angle = math.acos((b ** 2 + c ** 2 - a ** 2) / (2 * b * c)) * 57

    # ignore angles > 90 and ignore points very close to convex hull(they generally come due to noise)
    if angle <= 90 and d > 30:
        # add finger
        l += 1

# add finger
l += 1
```

Доколку неможеме да најдеме никакви дефекти функцијата терминира. Но ако најдеме дефекти ги листаме. И за секој дефект наоѓаме три точки почетокот и крајот на конвексот како и длабочината. За да минимизираме грешки и полесно да ги најдеме дефектите кои се поврзани со прстите го одредуваме аголот на најдлабоката точка во конвексот во однос на почетната и крајната точка на конвексот. Доколку аголот е помал од 90 степени тогаш знаеме дека дефектот е прст и го инкремнитираме бројачот за прсти.

```
    # print corresponding gestures which are in their ranges
    # if it has one finger
    if l == 1 and arearatio < 12:
        # display "Rock" text
        cv2.putText(frame, 'Rock', (0, 50), font, 2, (0, 0, 255), 3, cv2.LINE_AA)
        # add to the rock counter
        rock_count= rock_count + 1
    # if it has two finger
    elif l == 2:
        # display "Scissors" text
        cv2.putText(frame, 'Scissors', (0, 50), font, 2, (0, 0, 255), 3, cv2.LINE_AA)
        # add to the scissors counter
        scissors_count = scissors_count + 1
    # if it has five finger
    elif l == 5:
        # display "Paper" text
        cv2.putText(frame, 'Paper', (0, 50), font, 2, (0, 0, 255), 3, cv2.LINE_AA)
        # add to the paper counter
        paper_count = paper_count + 1
    # everything other then 1,2 or 5 fingers is an error
    else:
        # display "Error" text
        cv2.putText(frame, 'Error', (10, 50), font, 2, (0, 0, 255), 3, cv2.LINE_AA)

    # when the the first counter to reach 40 that move will be played by the user, before returning the move the counters will be reset
    if paper_count > 40:
        paper_count = 0
        rock_count = 0
        scissors_count = 0
        return "PAPER"
    elif scissors_count > 40:
        paper_count = 0
        rock_count = 0
        scissors_count = 0
        return "SCISSORS"
    elif rock_count > 40:
        paper_count = 0
        rock_count = 0
        scissors_count = 0
        return "ROCK"
```

Одкога ќе го најдеме бројот на прсти потребно е да одлучиме која форма ја прави корисникот.

Доколку откриеме еден дефект но проценотот на рака со конфексна обвивка е мал тогаш знаеме дека корисникот покажува камен. Доколку откриеме два или пет прста тогаш знаеме дека корисникот покажува ножички или лист. Доколку не исполниме ни еден од наведените услови тогаш преикжуваме ерор.

За да има време корисникот да ја прикаже својата форма имплементираме бројачи кои овозможуваат корисникот да прикаже форма во регионот од интерес па и дури да ја смени.

Кога бројачот ќе стигне до 40 формата се заклучува и играта продолжува.

За да има прегледност корисникот формата што алгоритамот ја открива се прикажува во горниот лев агол со големи црвени букви.

Како компјутерот да направи свој потег

Алгоритамот за правење на потег е доста едноставен. Алгоритмот ја користи функцијата за генерирање на рандом број и со користење на едноставна математика се одлучува за потег.

Во играта имплементирано новоа на тежина. Корисникот на почетокот од играта го одбира нивото на кое сака да игра. Секое ниво си има посебна функиција со одредени вреојатности на исход. Доколку нивото на тежината е лесно тогаш корисникот има поголема шанса да победи или да изедначи, но доколку нивото кое што го избрал е тежко тогаш корисникот може само да изедначи или изгуби.

За да ја имплементираме оваа функционалнот прво декларираме променлива game_mode која ќе држи број кој ја означува тежината на играта.

```
# Difficulty mode
game_mode = None
```

Потоа му ги прикажуваме опциите на корисникот и бараме влез од него. Доколку корисникот внесе нешто што не е број од 1 до 3 тогаш ги птинтаме опциите повторно и пак бараме внес од него.

```
# Display the difficulty modes and wait for the user to choose one
while(True):

    # the difficulty modes
    print("\nChoose a level of difficulty:\n"
          "1. Easy\n"
          "2. Medium\n"
          "3. Hard\n")

    # get the user input
    game_mode = int(input())

    # check if the user input is correct
    if game_mode <= 3 and game_mode > 0:
        break
```

По одредувањето на нивото на тежина во главната функција(main) чекаме корисникот да направи потег и декларираме променлива pc_choice која ќе го содржи потегот на компутерот.

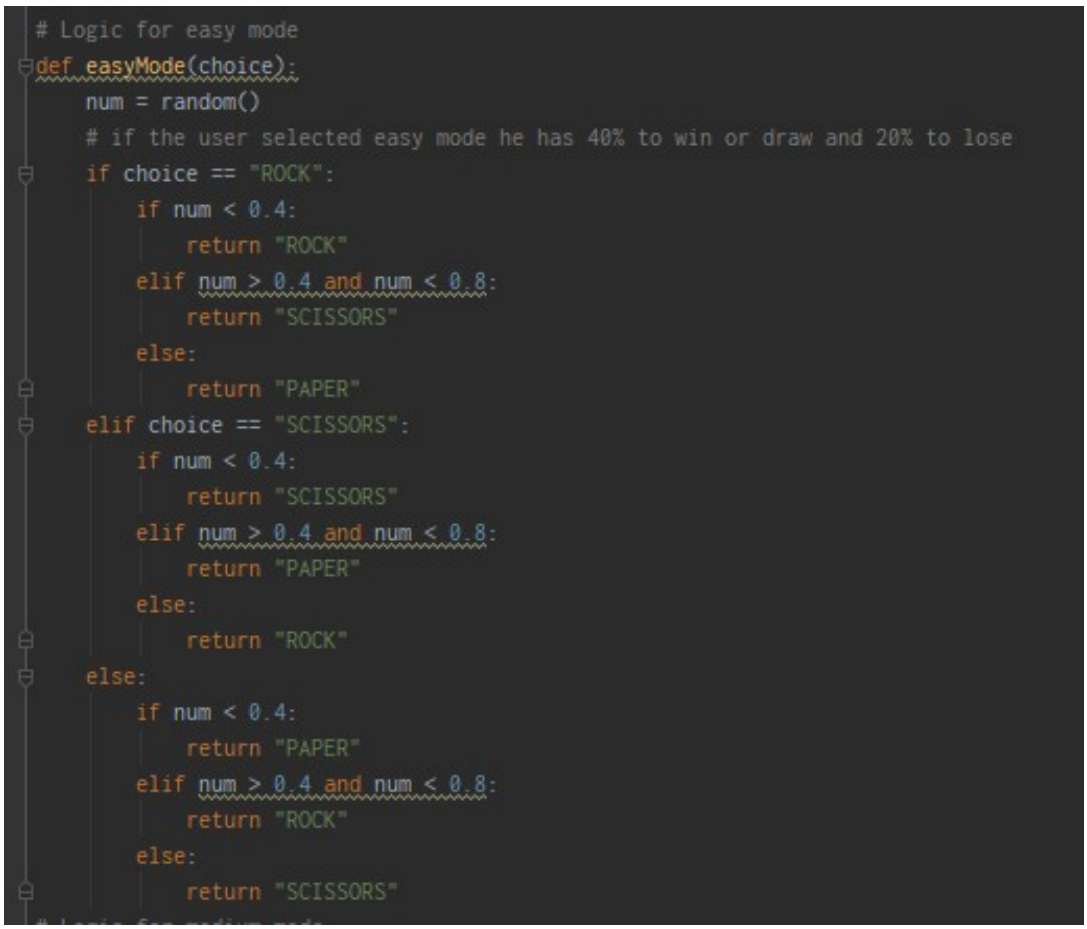
```
# check if the match is on or is in timeout
if play:
    # count the number of fingers and get the shape (rock, paper or scissors)
    choice = count_fingers(segmented, clone)

# var that holds the computers move
pc_choice = ""

# check if the match is on and if the user has made a move
if choice != None and play:
    # implement easy mode logic
    if game_mode == 1:
        pc_choice = easyMode(choice)
    # implement medium mode logic
    elif game_mode == 2:
        pc_choice = mediumMode(choice)
    # implement hard mode logic
    else:
        pc_choice = hardMode(choice)
    # find the result of the match
    analyse(pc_choice, choice)

    # set the game on timeout
    play = False
    # reset the hand var
    hand = None
```

Доколку корисникот има направено потег и играта не е во тајмаут се користи глобалната променлива game_mode за да се одреди која функција ќе се изврши за да се добие подег од компјутерот. По одредување на потег се анализираат потезите и се одредува резултатот. Играта ја ставаме во тајмаут за да можеме го прикажеме резултатот на корисникот и се брише сегментацијата на раката.

Сликата подоле опишива како функционира логиката за донесување на потег доколку корисникот го има изберено лесното ниво на тежина. Како аргумент оваа функција го зима choice кој го преставува потегот на корисникот. Со користење на овој аргумент можеме да видиме кои форми би придеонеле до победа, изедначување или пораз. Со ова ние можеме да ги контролираме веројатностите за исходот наместо целиот процес да е целосно рандомизиран. Истот така кеираме променлива num која претставува рандом број од 0 до 1.

Во оваа имплемнтација корисникот ќе има 40% да победи, 40% да изедначи и 20% да изгуби.

Другите имплементации за средното и тешкото ниво на тежина се исти само со различни шанси. Средното ниво има 33% да победи, 33% да изедначи и 33% да изгуби. Додека пак тешкото ќе има 50% за се изедначи и 50% корисникот да ја изгуби рундата.



```
# Logic for easy mode
def easyMode(choice):
    num = random()
    # if the user selected easy mode he has 40% to win or draw and 20% to lose
    if choice == "ROCK":
        if num < 0.4:
            return "ROCK"
        elif num > 0.4 and num < 0.8:
            return "SCISSORS"
        else:
            return "PAPER"
    elif choice == "SCISSORS":
        if num < 0.4:
            return "SCISSORS"
        elif num > 0.4 and num < 0.8:
            return "PAPER"
        else:
            return "ROCK"
    else:
        if num < 0.4:
            return "PAPER"
        elif num > 0.4 and num < 0.8:
            return "ROCK"
        else:
            return "SCISSORS"
```

По добивање на потег од компјутерот, алгоритамот ги анализира двата потези и го одредува резултатот од рундата. Ова го правиме со функцијата analyse која прима два аргументи

pc_choise – потегот на компјутерот

choise – потегот на корисникот

Алгоритамот е многу едноставен, користи многу услови за да го одреди резултатот. Кога ке се исполни некој од условите возависност од дали е победа, изедначување или пораз соодветните глобални променливи за резултат се инкрементираат и потоа се претставуваат на корисникот. По инкремнтирањето се враќа конечниот резултат од рундата.

```python
# Find the result of the match
def analyse(pc_choice, choice):
    global wins, draws, loses, result

    # check if it is a draw
    if choice == pc_choice:
        # update score
        draws = draws + 1
        # get the final result
        result = "DRAW"
    # check if it is lost
    elif choice == "ROCK" and pc_choice == "PAPER":
        # update score
        loses = loses + 1
        # get the final result
        result = "LOST"
    # check if it is won
    elif choice == "ROCK" and pc_choice == "SCISSORS":
        # update score
        wins = wins + 1
        # get the final result
        result = "WON"
    # check if it is won
    elif choice == "PAPER" and pc_choice == "ROCK":
        # update score
        wins = wins + 1
        # get the final result
        result = "WON"
    # check if it is lost
    elif choice == "PAPER" and pc_choice == "SCISSORS":
        # update score
        loses = loses + 1
        # get the final result
        result = "LOST"
    # check if it is won
    elif choice == "SCISSORS" and pc_choice == "PAPER":
        # update score
        wins = wins + 1
        # get the final result
        result = "WON"
    # check if it is lost
    elif choice == "SCISSORS" and pc_choice == "ROCK":
        # update score
        loses = loses + 1
        # get the final result
        result = "LOST"
```

Кога ќе го одредиме финалниот резултат алгоритамот продолжува во главната(main) функција каде проверуваме дали играте е во тајмаут. Доколку играта е во тајмаут се преверува резултатот во зависност од резултатот се појавува соодветна порака на средина на екранот за да го извезтиме корисникот. Времето на прикажувањето на пораката го

контролираме со counter променливата таа е почетно поставена на 150 тоа значи дека пораката ќе се прикажива на екранот и играта ќе биде во тајмаут за 150 тикови или фрејмови.

Вистинското време на приказ на пораката зависи од перформансите на уредот од кој се извечува играта.

Кога counter променливате ќе стигне до 0 новата рунда почнува, играта излага од тајмоут и counter-от се ресетира на 150.

И на крај го ажурираме глобалниот резултат.

```
# check if the match is no on
if not play:
    # check if the user won
    if result == "WON":
        # display victory message
        cv2.putText(clone, "YOU WON", (250, 300), font, 1, (0, 255, 0), 2,
                cv2.LINE_AA)
    # check if the user lost
    elif result == "LOST":
        # display defeat message
        cv2.putText(clone, "YOU LOST", (250, 300), font, 1, (0, 0, 255), 2,
                    cv2.LINE_AA)
    # check if the user drew
    elif result == "DRAW":
        # display tie message
        cv2.putText(clone, "DRAW", (250, 300), font, 1, (255, 0, 0), 2,
                    cv2.LINE_AA)
    # count down till the next match
    counter = counter - 1
    # check if the timeout has passed
    if counter == 0:
        # start the next match
        play = True
        # reset timeout counter
        counter = 150
# show score
cv2.putText(clone, "W: {} D: {} L: {}".format(wins, draws, loses), (0, 500), font, 1, (0, 0, 255), 2,
            cv2.LINE_AA)
```

Додатоци

За подобро искуство на корисникот имплементирав некои додатоци.Како информации и инструкции за како да се користи играта и како да се добијат најдобри резултати при сегментацијата на раката.

```
# Display the instructions and wait for user confirmation
while(True):

    # the instructions
    print("Firstly you will be asked to choose a difficulty mode.\n"
          "After choosing a mode a window will pop up it will show your camera feed and a green square.\n"
          "For the first 2-3 seconds don't make any movement inside the green square.\n"
          "After 3 seconds put your hand in the green square in a form of a rock, paper(fingers wide apart) or scissors.\n"
          "Keep your hand there until you see the result of the match in center of the window.\n"
          "After a couple of seconds the result message will be cleared and you can play another match.\n"
          "You can not play the game while the result message is displayed.\n"
          "Try to play the game in a room with good lighting and make sure that green square is filming a single colored background without any objects.\n"
          "To exit the game press 'q' on your keyboard.\n"
          "Do you understand?(yes)\n")
    # the user input
    tmp = input()

    # Check if input is correct
    if( tmp == "yes"):
        break
```

Корисникот мора да внесе ‘yes’ за да продолжи со играта. Доколку не внесе ‘yes’ инструкциите повторно ќе се испечатат и од корисникот повторно ќе биде побарано да впише ‘yes’.

Последниот додаток овозможува корисникот да ја изгаси играта во било кое време само со притискање на ‘q’

```
    # if the user pressed "q", then stop looping
    if cv2.waitKey(10) == ord("q"):
        print("Goodbye!")
        break
```

Потребни библиотеки за играта се

```
import cv2
import imutils
import math
from random import random
```