Przechwytywanie wideo na Raspberry Pi – sprawozdanie.

Naszym zadaniem było zaimplementowanie prostego programu do nagrywania obrazu, który byłby w bardzo prosty sposób modyfikowalny. Miało to umożliwiać łatwe rozszerzanie go o nowe funkcjonalności.
W implementacji posłużyliśmy się językiem Python, zważywszy na to, że jest preinstalowany na Raspbianie, a jego kod jest łatwy do zrozumienia.

Do samego przechwytywania obrazu użyliśmy biblioteki opencv, która zapewnia zaimplementowaną obsługę urządzeń nagrywających. Do modyfikacji obrazu użyliśmy numpy i opencv. Informację o filtrach, które zaimplementowaliśmy zaczerpnęliśmy ze strony opencv.org i wikipedii.
Obraz przechwytywany z użyciem powyższych bibliotek to 3 wymiarowa tablica, których wartości odpowiadają kolorom pixeli na obrazie(długość * szerokość * 3). Obraz rozdzielany jest na 3 warstwy, każda z nich odpowiada za jeden z trzech podstawowych kolorów. Po scaleniu tworzą obraz, który wyświetlamy w aplikacji.
Dzięki numpy możemy wydajnie działać na tych macierzach i modyfikować je dowolnie pod użycie filtrów.

Opis użycia, instrukcja do instalacji biblioteki opencv i krótka notka o programie znajdują się na repozytorium github:

Opis zaimplementowanych filtrów:

sobel_edge_det(self, frame, x, y)

```
def sobel_edge_det(self, frame, x, y):
    gray_scale = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    sobel = cv2.Sobel(gray_scale, cv2.CV_64F, x, y, ksize=5)
    matrix_fit = np.reshape(sobel, sobel.shape + (1,))
    return matrix_fit
```

Detekcja krawędzi za pomocą operatora sobela - jest to operator dyskretnego różniczkowania pozwalający na aproksymację pochodnych kierunkowych intensywności obrazu w ośmiu kierunkach co 45 stopni.

Aproksymacja pochodnych kierunkowych polega na konwolucji/splocie (http://www.songho.ca/dsp/convolution/convolution2d_example.html)
macierzy z obrazem z macierzą jądra*. Operacja splotu wyznacza nam estymatę odpowiedniej pochodnej cząstkowej
np. (dla 0 stopni względem osi X dla 90 stopni względem osi Y)

W użytej detekcji krawędzi możemy wyspecyfikować parametrami x,y podając wartości 0 lub 1 które macierze wykorzystamy do detekcji.
np. gdy x=1 y=0 będziemy wykrywać jedynie krawędzie poziome.
W przypadku użycia obu macierzy dla każdej wartości w macierzy wyliczamy "długość wektora" za pomocą wzoru:

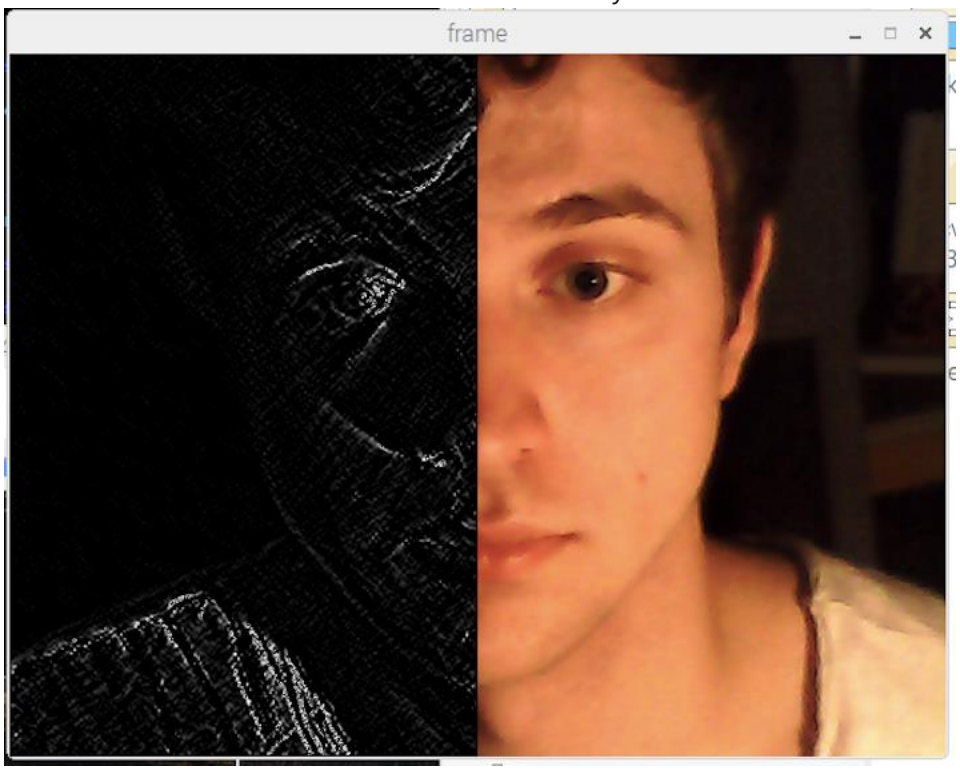
$$\mathbf{G} = \sqrt{\mathbf{G}_x^{\,2} + \mathbf{G}_y^{\,2}}$$

Możemy również wyliczyć kierunek kąt ze wzoru:

$$\Theta = \operatorname{atan}\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Standardowe macierze jądra Sobela mogą produkować znaczne niedokładności, użyta przez nas biblioteka opencv radzi sobie z takowymi nie dokładnościami

X=1 y=1



źródło:
opencv sobel edge detection

laplacian_edge_detecion(self, frame):

```
    return matrix_fil

def laplacian_edge_detecion(self, frame):

    return cv2.Laplacian(frame, cv2.CV_64F)
```

Detekcja krawędzi za pomocą operatora Laplace'a jest dużo bardziej wrażliwa na szum niż inne metody detekcji ze względu na to, że aproksymuje pochodne
drugiego stopnia.

Aby otrzymać lepsze efekty przed wykorzystaniem operatora Laplace'a można wykorzystać jakąś metodę usuwania szumu np. rozmycie gaussowskie.
Aby otrzymać podobny efekt wizualny (obraz czarno biały) należy na wykorzystywany obraz zaaplikowac filtr szarości.

W naszym przypadku nie użyliśmy wyżej wymienionych metod jedynie operatora la'placea, który również polega na konwolucji/splotu macierzy obrazu z macierzą jądra operatora Laplace'a:

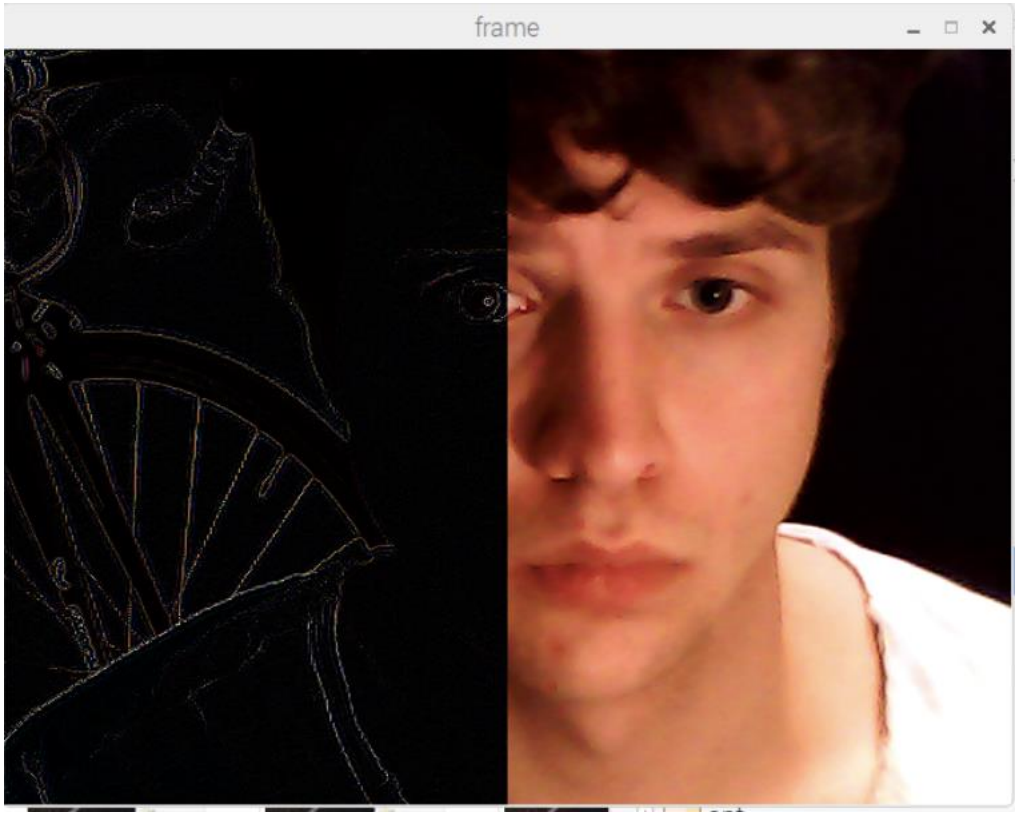
0	-1	0
-1	4	-1
0	-1	0

The laplacian operator

-1	-1	-1
-1	8	-1
-1	-1	-1

The laplacian operator (include diagonals)

The kernel for the laplacian operator



źródło:
opencv2 Laplace operator

canny_edge_det(self, frame, min, max):

```
def canny_edge_det(self, frame, min, max):
    canny = cv2.Canny(frame, min, max)
    return np.reshape(canny, canny.shape + (1,))
```

detekcja krawędzi za pomocą metody Canny dąży do określenia optymalnego algorytmu detekcji krawędzi. Optymalnego tzn. spełniającego 3 założenia:

- *dobrą detekcję* – algorytm powinien wykryć tak dużo krawędzi jak to tylko możliwe.
- *dobre umiejscowienie* – oznaczona krawędź powinna być jak najbliżej rzeczywistej krawędzi na obrazie.
- *minimalną odpowiedź* – konkretna krawędź powinna być oznaczona tylko raz, i jeśli tylko możliwe, zakłócenia (szum) w obrazie nie powinny tworzyć fałszywych krawędzi.

Algorytm działa według następujących kroków:

- **Redukcja szumu filtrem Gaussa**.
  Przykładowe jądro Gaussowskie o rozmiarze 5, którego moglibyśmy użyć :

$$K = \frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

- **Znalezienie gradientu intensywności(intensity gradient)**
  Aplikujemy następujące maski konwolucji jak w przypadku Sobel'a:

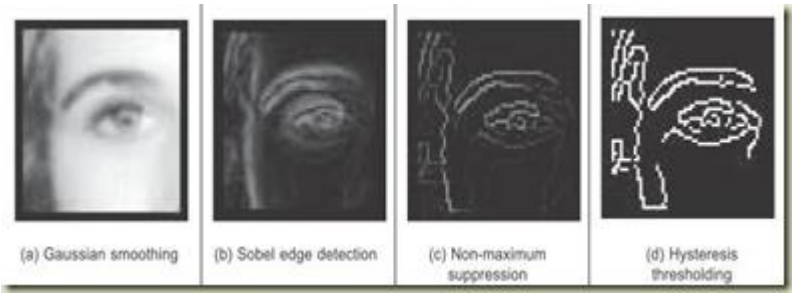
$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$
$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

  Następnie wyliczamy gradient i kierunek za pomocą wzorów:

$$G = \sqrt{G_x^2 + G_y^2}$$
$$\theta = \arctan(\frac{G_y}{G_x})$$

  kierunek(kąt) jest zaokrąglany do jednej z 4 możliwych wartości 0,45,90,135

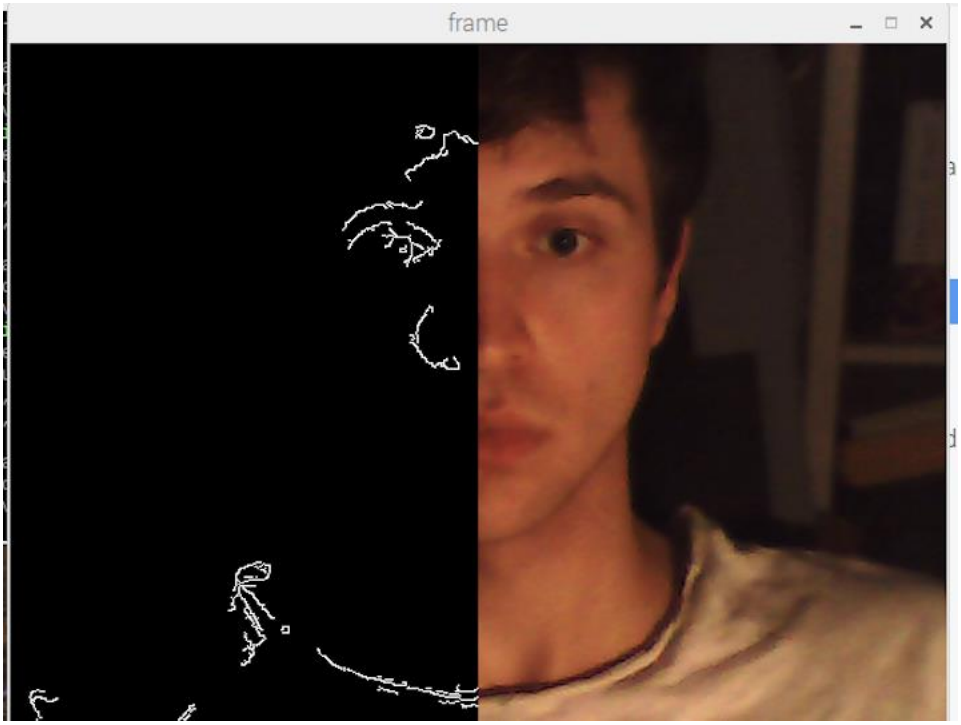
- Kolejnym krokiem jest (**non-maximum suppresion**) czyli specyficzne pociemnianie krawędzi poprzez usunięcie pixeli podejrzewanych o nienależenie do naszej krawędzi w wyniku czego pozostają jedynie cieńkie krawędzie.



http://weisu.blogspot.com/2009/05/

- **Ostatnim krokiem jest progowanie z histerezą**
  progowanie ma na celu usunięcie nieistotnych krawędzi,
  które mają nachylenie (stromość) poniżej ustawionego progu.
  Progowanie z histerezą powoduje, że do już wykrytych krawędzi są dołączane następne piksele mimo spadku nachylenia, aż do osiągnięcia dolnego progu wykrywania.
  Takie postępowanie zapobiega dzieleniu krawędzi w miejscach słabszego kontrastu.

  min=50 max=200



źródło:
opencv2 canny edge detection

threshold_gauss(self, frame, block_size, c, max_value=255,
adaptive_method=cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
threshold_type=cv2.THRESH_BINARY):

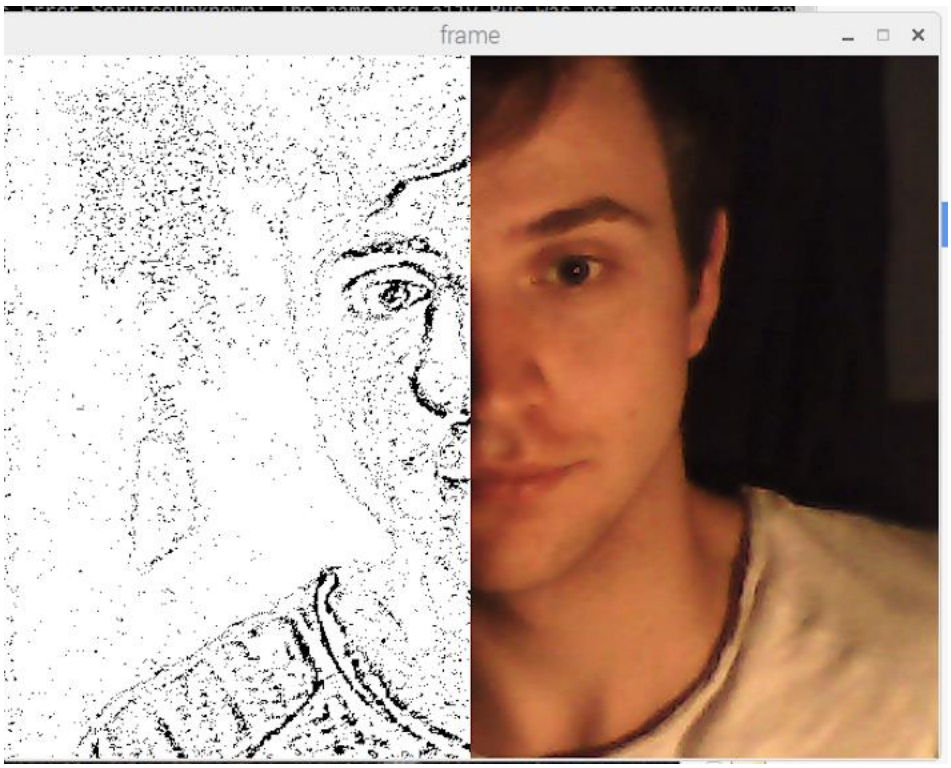
```
def threshold_gauss(self,
                    frame,
                    block_size,
                    c,
                    max_value=255,
                    adaptive_method=cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                    threshold_type=cv2.THRESH_BINARY):

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    gauss = cv2.adaptiveThreshold(gray, max_value, adaptive_method, threshold_type, block_size, c)
    matrix_fit = np.reshape(gauss, gauss.shape + (1,))
    return matrix_fit
```

Rozmycie Gaussa(filtr Gaussa) - popularne rozmycie obrazu za pomocą funkcji Gaussa często wykorzystywane w grafice komputerowej:

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

Block_size=11
C=2

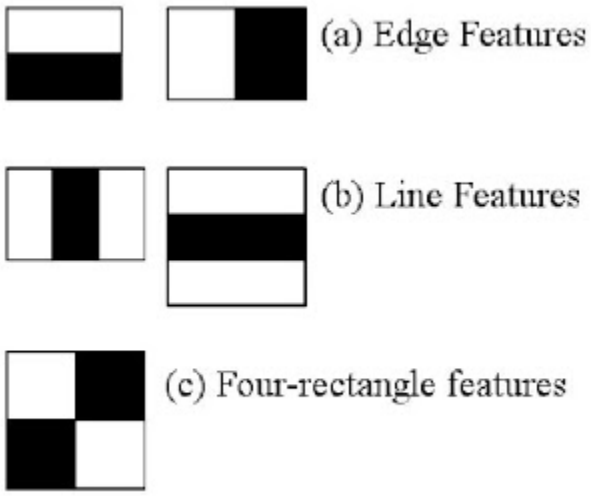


face_detection(self, frame):

```
def face_detection(self, frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    faces = self.face_haar_cascade.detectMultiScale(gray, 1.3, 5)
    for (x, y, w, h) in faces:
        cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
    return frame
```

Detekcja twarzy przy użyciu “Haar Cascades” jest to podejście machine learningowe, w którym “cascade function” jest trenowana na wielu obrazkach pozytywnych oraz negatywnych i następnie wykorzystana do wykrywania obiektów na innych obrazach.

W naszym przypadku wykrywamy twarze więc do wytrenowania algorytmu pozytywnymi obrazami będą takie zawierające twarze natomiast negatywnymi obrazy nie zawierające twarzy.
Algorytm polega na wyekstrahowaniu z obrazów następujących właściwości haar (haar features)

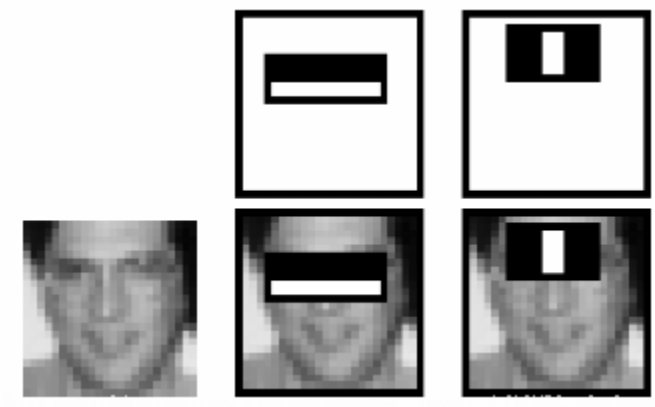


(docs.opencv.org/3.3.0/d7/d8b/tutorial_py_face_detection)

Każda właściwość jest pojedynczą wartością liczoną przez odjęcie sumy wartości w białej części od sumy wartości w części czarnej.

Można zauważyć, że w rezultacie właściwości tych dla każdego rozmiaru i każdej pozycji jest bardzo dużo (ponad 160 000 właściwości dla obrazku rozmiaru 24x24) żeby sobie z tym poradzić stosuje “Summed-area table” inaczej też zwaną “Integral images” służy ona do szybkiego obliczania sum wartości w prostokątnych podmacierzach zadanej macierzy.

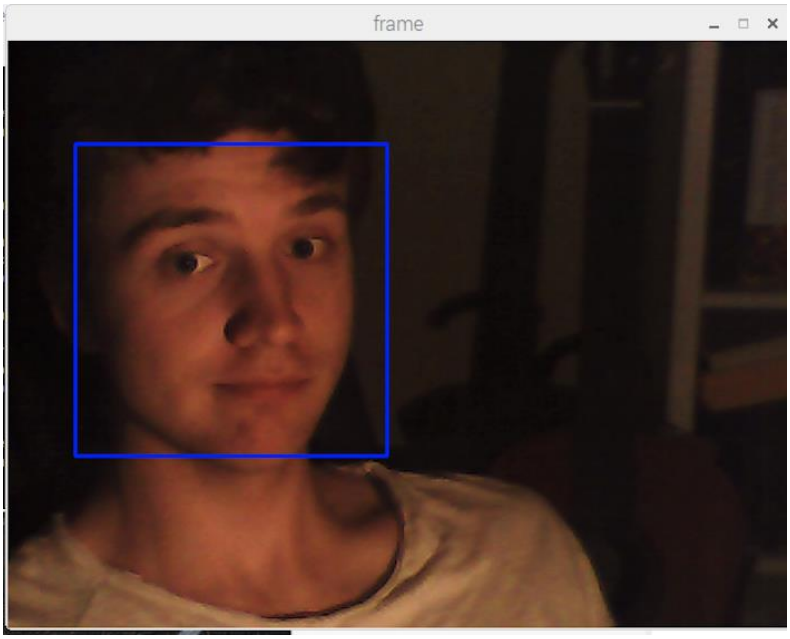
Możemy też się domyślić że znaczna większość z nałożonych właściwości okaże się bezużyteczna



(docs.opencv.org/3.3.0/d7/d8b/tutorial_py_face_detection)

Właściwość na powyższym obrazku są przydatne uwzględniają np. przerwę między brwiami.
Lecz gdyby umieścić te same maski w innych miejscach na przykład na policzkach okazałyby się bezwartościowe.
Do filtracji nieprzydatnych właściwości wykorzystywany jest algorytm **Adaboost**.
Algorytm **Adaboost** pozwala na znaczne zredukowanie liczby właściwości np. z 160 000 do 6000 a prace naukowe mówią, że wystarczy nawet jedynie 200 właściwości do osiągnięcia 95% powodzenia.
Więc po zredukowaniu liczby właściwości wystarczy zaaplikować wszystkie na obraz i stwierdzić czy jest twarzą, lecz aplikowanie wszystkich 6000 właściwości byłoby bardzo nieefektywne więc stosowane są heurystyki żeby sobie z tym poradzić a mianowicie:

Najpierw sprawdzamy czy część obrazka, którą pokrywa właściwość jest w ogóle częścią twarzy bo jak wiemy większa część obrazków nimi nie będzie. Właściwości, które przejdą ten test negatywnie od razu odrzucamy bez dalszej analizy.
Następnie wprowadzony jest koncept kaskady klasyfikatorów (“Cascade of filters”) czyli właściwości są podzielone na grupy i etapy następnie aplikujemy każdy etap jeśli obrazek nie przejdzie któregoś etapu odrzucamy go.
W naszym przykładzie z 160 000 właściwości zredukowanymi do 6000 autor opowiada, że powstało u niego 38 etapów po kolejno 1,10,25,25,50 właściwości w pierwszych pięciu etapach. W rezultacie z całych 6000 właściwości średnio wystarczyła analiza 10 na sub-okienko obrazka.



źródło:
opencv2 face detection

cartoon_filter(self, frame):

```
def cartoon_filter(self, frame, downscale=2, bilateral_steps=5):
    for _ in range(downscale):
        frame = cv2.pyrDown(frame)
    for _ in range(bilateral_steps):
        frame = cv2.bilateralFilter(frame, d=9, sigmaColor=9, sigmaSpace=7)
    for _ in range(downscale):
        frame = cv2.pyrUp(frame)
    gray_scale = cv2.cvtColor(frame, cv2.COLOR_RGB2GRAY)
    edge = cv2.adaptiveThreshold(cv2.medianBlur(gray_scale, 7), 255,
                                 cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 2)
    return cv2.bitwise_and(frame, cv2.cvtColor(edge, cv2.COLOR_GRAY2RGB))
```

Do stworzenia efektu kreskówki posłużyliśmy się filtrem Bilateral, który potrafi wygładzić obszary obrazu zachowując przy tym ostrość krawędzi. Sam filtr matematycznie zdefiniowany jest następująco:

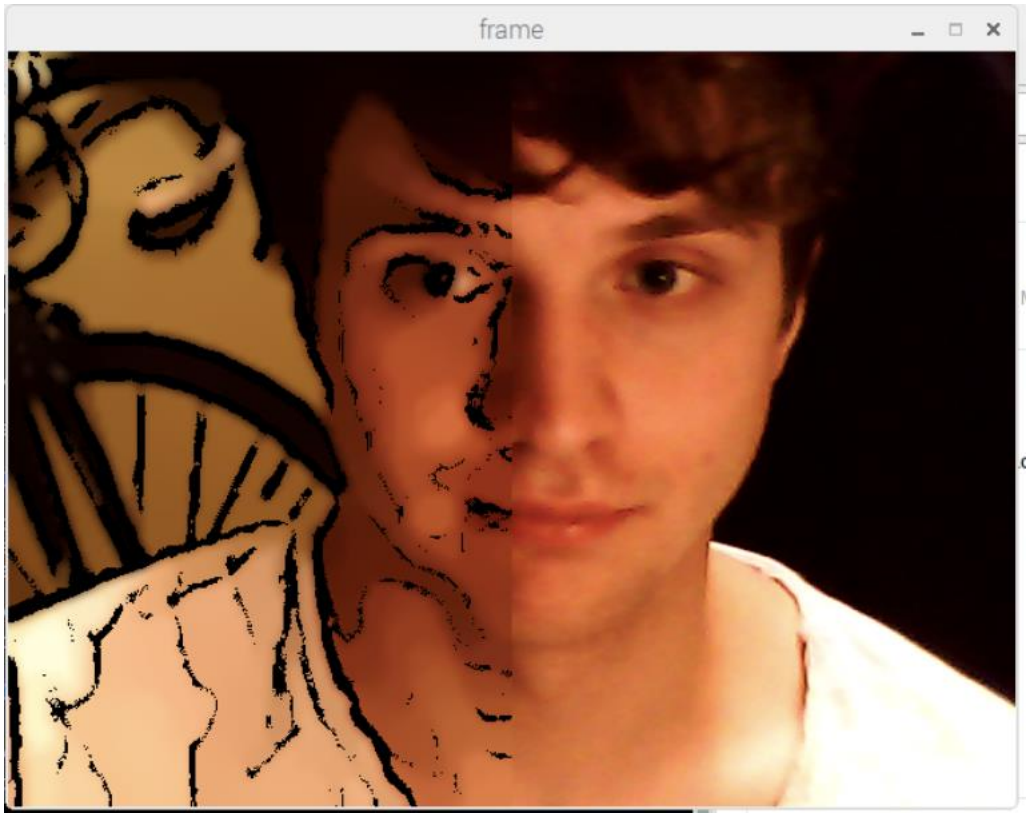
$$I^{\text{filtered}}(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) f_r(\|I(x_i) - I(x)\|) g_s(\|x_i - x\|),$$

, gdzie:
I^{filtred} - przefiltrowany obraz,
I - obraz przed filtracją,
X – koordynaty aktualnego pixela, który ma być zmodyfikowany,
Ω - obraz wycentrowany w x,
F_r – range kernel
G_s – spatial kernel

Niestety filtr sam w sobie jest dość ciężki, więc aby zastosować do na Raspberry na samym początku zeskalowaliśmy obraz w dół i następnie na tak zeskalowany obraz założyliśmy 5-krotnie mały filtr Bilaterela, który daję taki sam efekt jak pojedyncze nałożenie mocnego filtru z tym, że jest to mniej zasobożerne. Po wykonaniu przetwarzania zeskalowaliśmy ponownie zdjęcie do początkowych wartości.
Następnie wykonaliśmy redukcję szumów używając do tego median blura, delikatne rozmycie nic pozbawi nas efektu kreskówi, a usuniemy brzydko wyglądający szum na obrazie.

Median blur zastępuję dany pixel średnią wartością pixeli w małym sąsiedztwie, w naszym przypadku równym 7.

Następnie wykonaliśmy adaptive threshold, znany z poprzednich punktów. Dał nam on warstwę obrazy z wyszczególnionymi krawędziami. Na koniec scalamy obydwa obrazy.



źródło:
https://en.wikipedia.org/wiki/Bilateral_filter