



Projekt realizowany w ramach przedmiotu
Systemy Komputerowe w Sterowaniu i Pomiarach
kierunek: Informatyka
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

1 Wprowadzenie

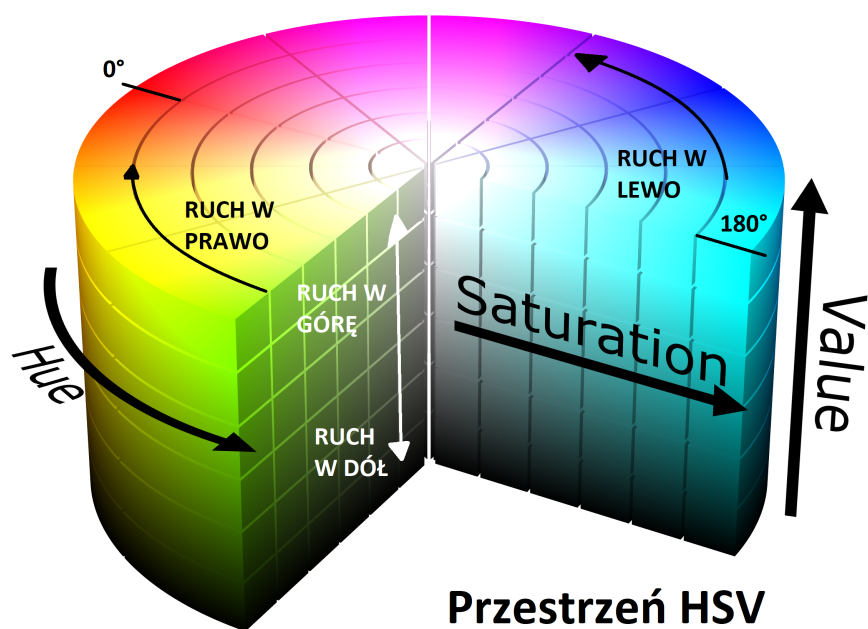
Celem projektu jest stworzenie szybko reagującego systemu czasu rzeczywistego przetwarzającego 4 gesty, dzięki któremu możliwe będzie sterowanie barwą oraz jasnością kwadratu znajdującego się na stronie.

1.1 Skład zespołu

Imię i Nazwisko	Nr albumu
Karolina Romanowska	304120
Michał Matak	304071

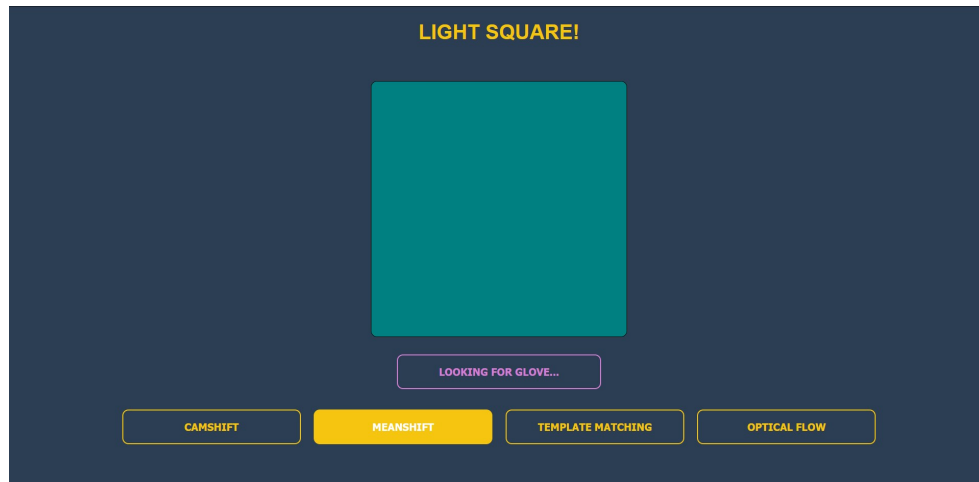
Tabela 1: skład zespołu

1.2 Opis funkcjonalności systemu



Rysunek 1: Wpływ kierunku ruchu na barwę

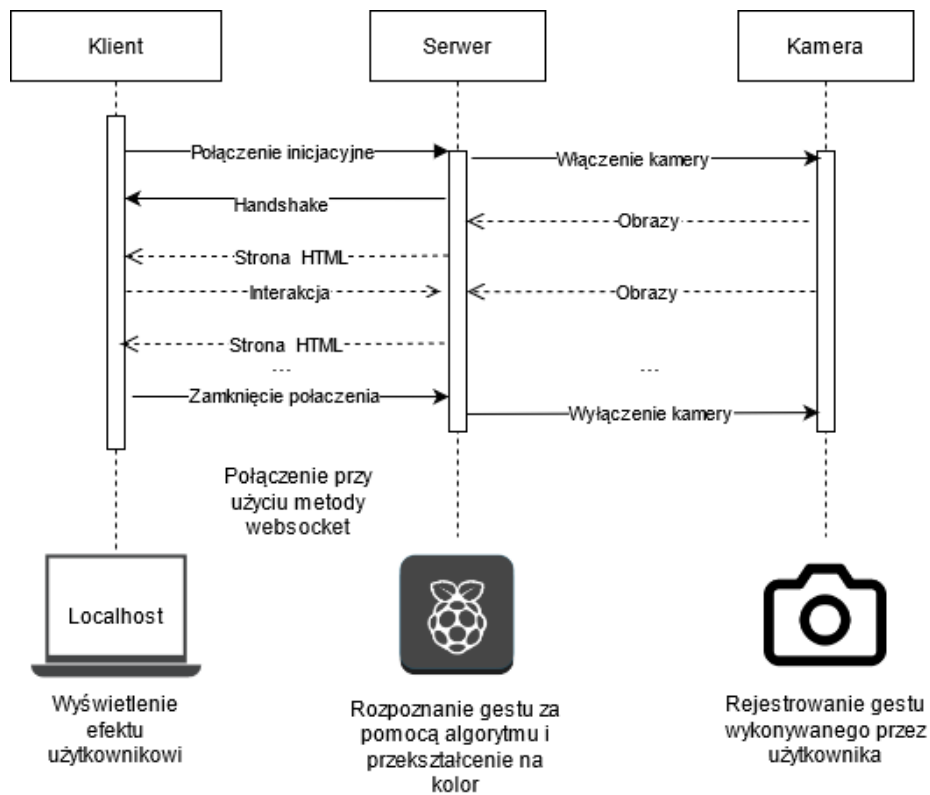
System ma za zadanie sterować barwą elementu znajdującego się na stronie poprzez wykonywanie gestów przed kamerą. Edytowany jest kolor będący w przestrzeni HSV. Na rysunku widoczny jest wpływ gestu na barwę. Podczas ruchu w górę zwiększana jest wartość V (Value) a kolor zwiększa swoją jasność. Odwrotnością tego jest ruch w dół, podczas którego zmniejszana jest wartość V, co powoduje zmianę koloru na ciemniejszą. Ruchy horyzontalne wpływają na wartość H (Hue) - odcień światła.



Rysunek 2: Widok klienta

Ponieważ każdy z zaimplementowanych algorytmów śledzących obiekt ma swoje wady i zalety, umożliwiliśmy użytkownikowi wybór wykorzystywanego algorytmu.

2 Schemat systemu



Rysunek 3: Schemat systemu

2.0.1 Opis poszczególnych elementów systemu

Podczas testów wykorzystano następujący sprzęt:

- Raspberry Pi 4B
 - Procesor - Broadcom BCM2711 quad-core 64-bitowy ARM-8 Cortex-A72 1,5 GHz
 - Pamięć RAM - 4GB
 - Karta microSD - 32GB
- Kamera TRACER HD WEB008
 - Typ sensora - CMOS
 - Rozdzielczość - 1280x720
 - Kompresja wideo - MJPEG
 - Interfejs - USB

Wykorzystane języki programowania oraz biblioteki:

- tracklib - moduł rozpoznawania gestów
 - OpenCV
 - Numpy
- serwer
 - Python
 - FastAPI
- klient
 - JavaScript
 - HTML
 - CSS

3 Opis zastosowanych algorytmów

3.1 Find Pink Glove

Aby poprawić efektywność działania algorytmów rozpoznawania gestu wprowadziliśmy algorytm znajdowania wyróżniającego się koloru. W przypadku naszej implementacji jest to kolor różowy. Sposób działania algorytmu:

- Stworzenie przedziału koloru przy pomocy funkcji `cv2.inRange()` i uzyskanie maski
- Następnie ta maska jest nakładana na obraz przechwycony z kamery
- Przy pomocy funkcji `cv2.findContours()` wydobywane są kontury elementu uwidocznionego poprzez nałożenie maski. Funkcja ta wykorzystuje algorytm autorstwa Satoshi Suzuki i innych opisany w źródle o tytule “Topological structural analysis of digitized binary images by border following. Computer Vision, Graphics, and Image Processing” z roku 1985
- Następnie wybierany jest największy rozpoznany obszar, który będzie wejściem dla algorytmów śledzących.

3.2 Meanshift (Algorytm zmiany średniej)

Algorytm, który iteracyjnie przesuwa środek punktów zainteresowań do punktu, w którym średnia punktów znajduje się w sąsiedztwie. W wyniku czego uzyskujemy trajektorie i możemy śledzić środek obiektu. Sposób działania algorytmu:

- Na wejściu algorytm otrzymuje początkową ramkę wideo oraz wyznaczony wcześniej początkowy obszar zainteresowania
- Wyznaczany jest histogram obiektu zainteresowań, który jest następnie normalizowany. Wykorzystywany jest on podczas przeprowadzania projekcji wstecznej. Histogram obrazu zainteresowania da prawdopodobieństwo, że dany piksel należy do określonego obszaru.
- Wykonywana jest projekcja wsteczna przy użyciu funkcji `cv2.calcBackProject()`. Polega ona na zmianie wartości piksela obrazu wejściowego na odpowiadającą mu wartość ze znormalizowanego histogramu. Dzięki temu otrzymujemy prawdopodobieństwo występowania wyznaczonego obiektu zainteresowania.
- Przy pomocy funkcji `cv2.meanShift()` wykonywany jest krok algorytmu polegający na przesunięciu punktu wejściowego do średniej punktów w otoczeniu danego punktu. Przekłada się to na przesuwanie się wyznaczonego przez nas punktu do obszaru o najwyższym prawdopodobieństwie. Zaimplementowany na podstawie algorytmu opublikowanego przez Garego Bradsky w opracowaniu "Computer Vision Face Tracking for Use in a Perceptual User Interface" z roku

Zalety	Wady
Krótki czas iteracji, wydajny. Wykorzystywany dla systemów czasu rzeczywistego. Idealny dla przestrzeni kontrastowych	Rozpoznawany obiekt musi mocno wyróżniać się z tła. Nie może zmienić rozmiaru (przybliżyć do kamery) ani orientacji.

Tabela 2: Pozytywne oraz negatywne algorytmu Meanshift

3.3 Camshift (Adaptacyjny algorytm zmiany średniej)

Rozszerzenie algorytmu Meanshift o możliwość zmiany orientacji czy rozmiaru. Sposób działania algorytmu:

- Na wejściu algorytm otrzymuje początkową ramkę wideo oraz wyznaczony wcześniej początkowy obszar zainteresowania.
- Jako że algorytm ten jest udoskonaleniem algorytmu zmiany średniej, ponownie wyznaczany jest histogram oraz wykonywana jest projekcja wsteczna.
- Przy pomocy funkcji `cv2.CamShift` wykonywany jest kolejny krok algorytmu. Krok ten polega na początkowym wywołaniu kroku algorytmu Meanshift. Następnie aktualizowany jest rozmiar okna $s = 2 * \sqrt{\frac{M_{00}}{256}}$ oraz obliczana jest orientacja najlepiej dopasowanej do obiektu elipsy

Zalety	Wady
Okno śledzące obiekt może zmieniać rozmiar	Nie może być wykorzystany do śledzenia skomplikowanych kształtów, tło nie może być różnorodne, powinno być jednolite.

Tabela 3: Pozytywne oraz negatywne algorytmu Camshift

3.4 Optical Flow (Przepływ optyczny)

Wybiera krawędzie obiektu i śledzi ich wektor ruchu. Wyodrębnione punkty są przekazywane między kolejnymi klatkami aby zapewnić śledzenie tych samych obiektów. Sposób działania:

- Na wejściu algorytm otrzymuje początkową ramkę wideo oraz wyznaczony wcześniej początkowy punkt zainteresowania, który jest środkiem obszaru wyznaczonego przez FindPinkGlove()
- Następnie obraz jest konwertowany do skali szarości
- Ostatecznie wywoływana jest funkcja `cv.calcOpticalFlowPyrLK()` która to wylicza przepływ optyczny używając iteracyjnej, piramidalnej wersji algorytmu Lucas-Kanade. Przesunięcie jest tutaj określane przy pomocy metody gradientowej. Rozwinięcie algorytmu w piramidę polega na analizie obrazów z coraz większą rozdzielczością, a co za tym idzie z coraz większą liczbą detali. Dzięki temu przeciwdziałamy negatywnym skutkom wyjścia śledzonego obiektu za kadr.

Zalety	Wady
Dobrze działa nawet przy skomplikowanym tle, oraz przy szybkich ruchach	Kosztowny ze względu na skomplikowane obliczenia

Tabela 4: Pozytywne oraz negatywne algorytmu Optical Flow

3.5 Template Matching

Dostarczany jest algorytmowi szablon, który to jest następnie po kolei przykładany do kolejnych rzędów pikseli. Sposób działania:

- Na wejściu algorytm otrzymuje początkową ramkę wideo oraz wyznaczony wcześniej początkowy szablon
- Następnie podczas kolejnych kroków algorytmu wywoływana jest funkcja `cv.matchTemplate()`, która przesuwa szablon nad wejściową ramką wideo i porównuje oba fragmenty. Zwracany jest obraz, w którym każdy piksel którego wartość oznacza jak bardzo sąsiedztwo tego piksela pasuje do szablonu.

Zalety	Wady
Prosty w implementacji i użyciu. Radzi sobie z tymczasowym zniknięciem śledzonego obiektu.	Źle radzi sobie ze skomplikowanymi wzorcami.

Tabela 5: Pozytywne oraz negatywne algorytmu Template Matching

3.6 Gesture Classifier

Klasa implementująca funkcjonalność rozpoznawania gestów na podstawie lokalizacji N ostatnich punktów zainteresowań Sposób działania:

- Każdy z algorytmów przetwarzających zdjęcie w celu identyfikacji gestu wywołuje funkcję `classify_with_coords()/classify_with_points()`
- Funkcja ta bierze N ostatnich znalezionych obszarów zainteresowań, wybiera ekstrema i na ich podstawie wylicza ostatni kierunek przemieszczenia, następnie zwraca numer oznaczający gest.

Oznaczenia:

- 1 - ruch w dół
- 2 - ruch w górę
- 3 - ruch w prawo
- 4 - ruch w lewo

4 Opis wykonanych eksperymentów

Aby poznać możliwości i ograniczenia systemu, wybranych algorytmów oraz sprzętu przeprowadziliśmy eksperymenty polegające na pomiarze czasu danych czynności.

4.1 Pomiar szybkości kamery

Według producenta, nasza kamera podłączona do Raspberry działa z prędkością 25 klatek na sekundę. Aby sprawdzić czy dane te są prawdziwe oraz czy użyta biblioteka do obsługi lub obciążenie procesora albo wejścia/wyjścia wpływają na szybkość działania, napisaliśmy skrypt w pythonie, który mierzy tą szybkość i był odpalany wraz z programem *stress* dla różnych parametrów. Pomiar szybkości następuje poprzez wykonanie 120 zdjęć kamerką internetową i podzielenia 120 przez długość czasu wykonania. Aby mieć punkt odniesienia na początku wykonaliśmy test na nieobciążonej kamerce:

```
pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.087448596954346 seconds
Estimated frames per second : 23.587461910050404
```

Rysunek 4: Pomiar dla kamery przy nieobciążonym urządzeniu

Następnie na oddzielnym ekranie wykonaliśmy komendy (odpowiednio dla kolejnych testów) obciążające procesor:

```
sudo stress --cpu 10 --timeout 1000
sudo stress --cpu 100 --timeout 1000
sudo stress --cpu 1000 --timeout 1000
```

```

pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.086944103240967 seconds
Estimated frames per second : 23.589801178185983
pi@raspberrypi:~/SKPS---gesture-recognition $ screen -r 10277.stress
[detached from 10277.stress]
[detached from 10277.stress]
pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.087559700012207 seconds
Estimated frames per second : 23.586946802749473
pi@raspberrypi:~/SKPS---gesture-recognition $ screen -r 10277.stress
[detached from 10277.stress]
[detached from 10277.stress]
pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.08789324760437 seconds
Estimated frames per second : 23.585400510614466

```

Rysunek 5: Pomiar dla kamery przy obciążonym procesorze

Wytworzyliśmy także obciążenie na wejściu/wyjściu:

```
sudo stress --io 1000 --timeout 100
```

```

pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.087073087692261 seconds
Estimated frames per second : 23.589203050832857
pi@raspberrypi:~/SKPS---gesture-recognition $ █

```

Rysunek 6: Pomiar dla kamery przy obciążonym wejściu/wyjściu

I zarówno na procesorze jak i wejściu/wyjściu jednocześnie:

```
sudo stress --cpu 1000 --timeout 1000 & stress --io 1000 --timeout 1000
```

```

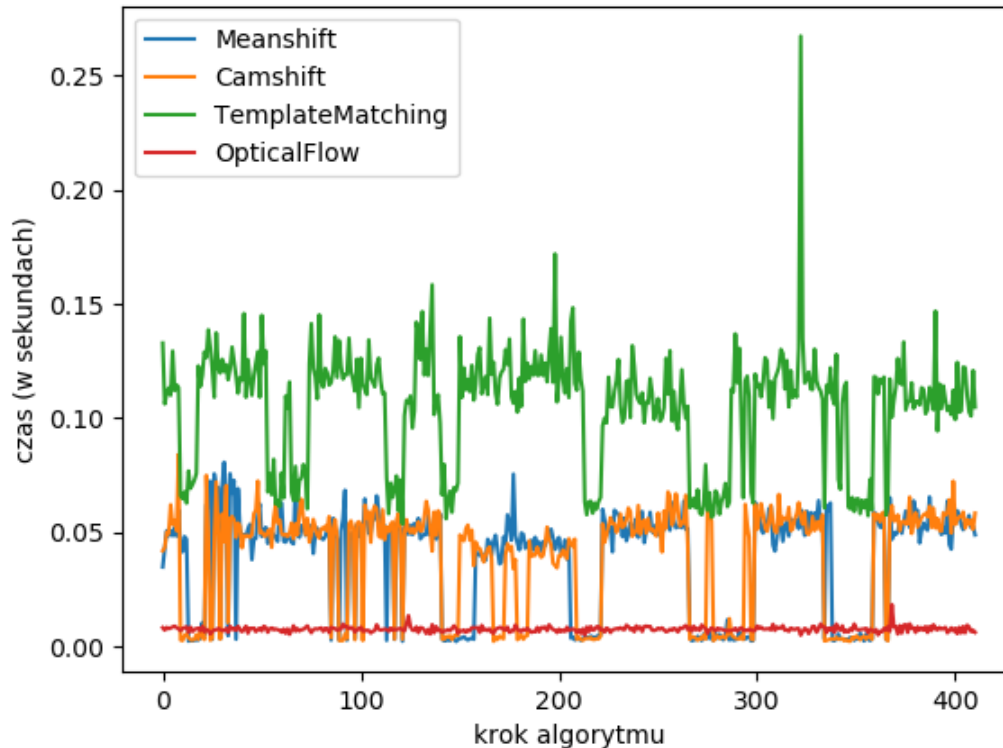
pi@raspberrypi:~/SKPS---gesture-recognition $ python3 test_camera_performance.py
Frames per second using video.get(cv2.CAP_PROP_FPS) : 25.0
Capturing 120 frames
Time taken : 5.086812734603882 seconds
Estimated frames per second : 23.59041039267678
pi@raspberrypi:~/SKPS---gesture-recognition $ █

```

Rysunek 7: Pomiar dla kamery przy obciążonym procesorze i wejściu/wyjściu

4.2 Pomiar czasu działania algorytmów

W naszym projekcie są dostępne 4 algorytmy wykrywania gestów. Aby porównać długość trwania kroku algorytmu i zorientować się ile ona orientacyjnie wynosi dla każdego algorytmu, napisaliśmy dekorator, który mierzy czas wykonania dowolnej funkcji i zapisuje go do pliku. Po udekorowaniu nim metody odpowiedzialnej za wykrywanie rękawiczki zmierzaliśmy czas jaki zajmuje wykonanie tej operacji. Algorytmy były wykonywane na kolejnych klatkach z nagranych filmu. Zebrane dane znajdują się na poniższym wykresie:



Rysunek 8: Czas działania wymienionych algorytmów

4.3 Pomiar minimalnej liczby klatek na sekundę potrzebnej do prawidłowego działania

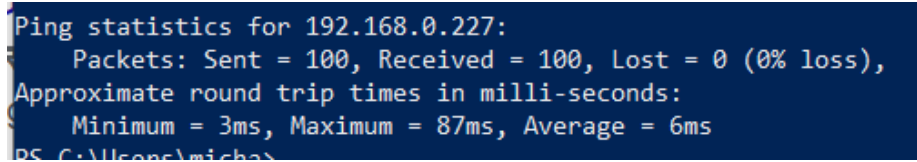
Aby określić ile klatek na sekundę musi dostawać każdy program postanowiliśmy uruchomić program na nagranych filmie, który składał się z 531 klatek i trwał 34 sekund (co daje średnio 15,61 klatek na sekundę), biorąc z niego co n -tą klatkę i wyświetlając ją wraz z wykrytym obrazem subiektywnie określiliśmy przy jakiej liczbie n algorytm przestaje dawać zadowalające rezultaty.

Algorytm	n
Meanshift	3
Camshift	2
Template Matching	15
Optical Flow	2

Tabela 6: maksymalny przeskok klatek

4.4 Opóźnienie transmisji

Poleceniem ping sprawdziliśmy opóźnienie transmisji, aby zobaczyć jaki czas może zostać poświęcony na połączenie.



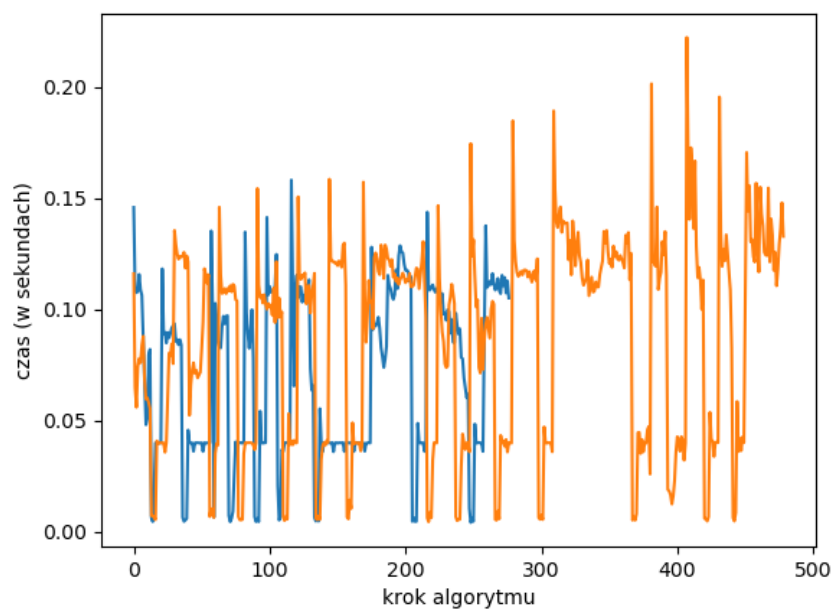
```
Ping statistics for 192.168.0.227:
    Packets: Sent = 100, Received = 100, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 3ms, Maximum = 87ms, Average = 6ms
PS C:\Users\micha>
```

Rysunek 9: efekt polecenia ping

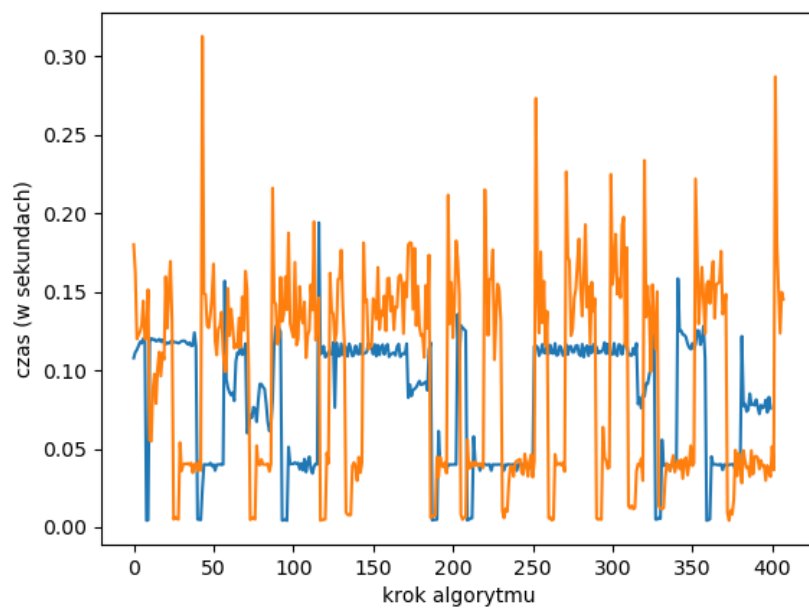
4.5 Pomiar czasu wykonania głównej pętli

Aby dowiedzieć się ile czasu zajmuje wykonanie głównej pętli programu za pomocą funkcji *time* mierzyliśmy czas wykonania pętli i wypisywaliśmy wyniki do pliku. Operacje taką wykonaliśmy dla każdego algorytmu. Zmiana algorytmu następowała poprzez wybór odpowiedniej opcji na stronie internetowej. Pomiary wykonaliśmy dla systemu zarówno obciążonego (kolor pomarańczowy na wykresie) jak i nieobciążonego (kolor niebieski na wykresie). Do obciążenia systemu użyliśmy komendy:

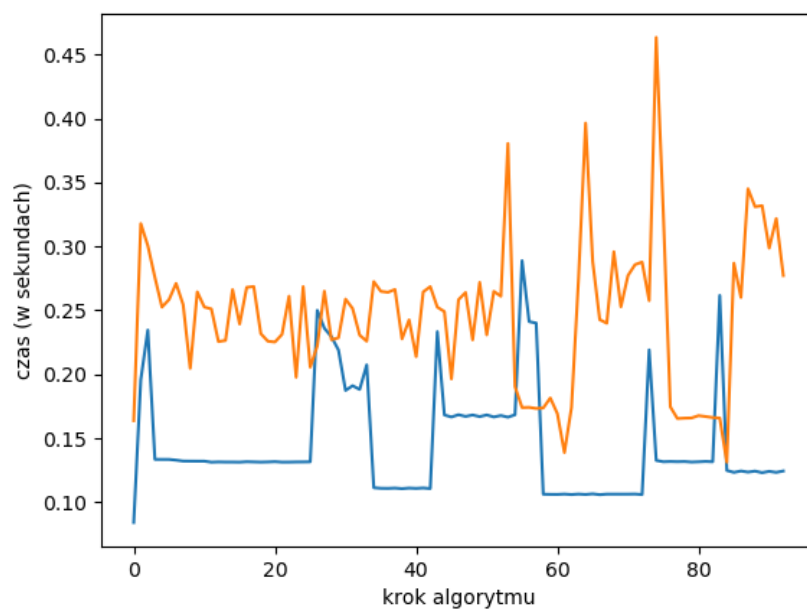
```
sudo stress --cpu 1000 --timeout 1000 & stress --io 1000 --timeout 1000
```



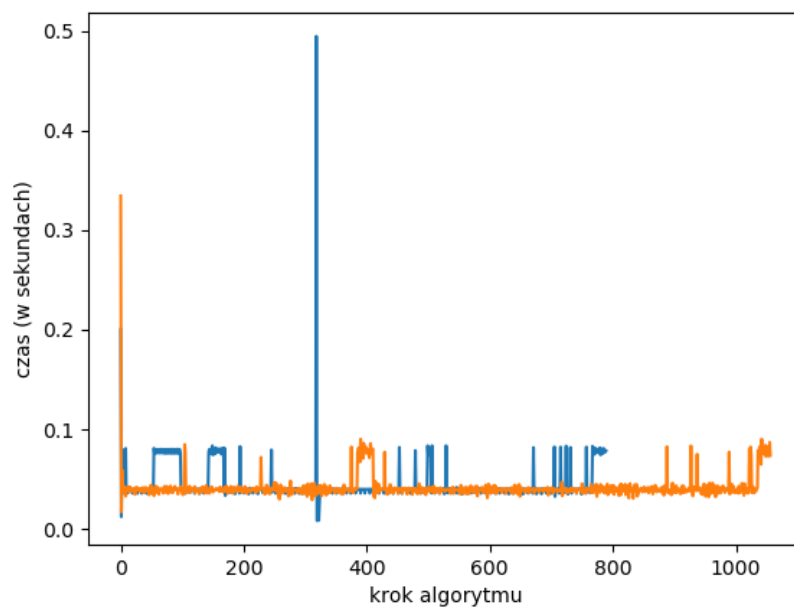
Rysunek 10: Pomiar czasu wykonania głównej pętli przy włączonym algorytmie *Meanshift*



Rysunek 11: Pomiar czasu wykonania głównej pętli przy włączonym algorytmie *Camshift*



Rysunek 12: Pomiar czasu wykonania głównej pętli przy włączonym algorytmie *Template Matching*



Rysunek 13: Pomiar czasu wykonania głównej pętli przy włączonym algorytmie *Optical flow*

5 Wnioski z wykonanych eksperymentów

Z przeprowadzonych eksperymentów możemy stwierdzić, że obciążenie systemu nie wpływa na szybkość działania kamery. Z kolei czas poświęcony na transmisję danych sięga kilku milisekund. Wiedząc, że średnia szybkość klatek filmu testowego wynosi 15 klatek na sekundę i przyjmując wcześniejsze oznaczenie n możemy oszacować ile powinna wynosić minimalna liczba klatek na sekundę dla algorytmu $(15, 61/n)$ oraz jak długo maksymalnie może trwać główna pętla w przypadku danego algorytmu $(1000ms \cdot \frac{n}{15,61})$. Patrząc na dane pochodzące z analizy głównej pętli możemy stwierdzić, że działając

Algorytm	Minimalna liczba klatek	Maksymalny czas trwania głównej pętli (w milisekundach)
Meanshift	5	200
Camshift	8	125
Template Matching	1	1000
Optical Flow	8	125

Tabela 7: wymagania systemu dla danych algorytmów

pod obciążeniem, nasz system będzie działał całkiem dobrze pod względem wydajnościowym, gdyż wszystkie algorytmy poza *camshiftem* mają dużo mniejszy czas niż wymagany przez oszacowanie. W przypadku *camshifta* średnia jest na granicy wymaganej wartości.

6 Zmiany projektowe

6.1 Migracja na płytkę Raspberry Pi

Połączenie części nad którymi oddzielnie pracowaliśmy okazało się nie możliwe. Wynikało to z powodu użycia biblioteki FastAPI - nowoczesnej i szybkiej, obsługującej ASGI. Z tego względu postanowiliśmy, za zgodą prowadzącego, przenieść system na płytkę Raspberry Pi z systemem Raspberry Pi OS lite arm64. Dodatkowo na płytce użyliśmy lepszej wersji biblioteki OpenCV - opencv-python-headless. Nie zawiera ona żadnych funkcjonalności GUI takich jak przykładowo - wyświetlanie obrazu.

6.2 Początkowa konfiguracja jądra i systemu w Buildroocie

Pierwotnie założenia projektu zakładały, że obraz systemu operacyjnego zostanie utworzony za pomocą buildroota i odpalony na maszynie wirtualnej emulowanej przez qemu. Jednym z pierwszych celów było wytworzenie obrazu systemu, który pozwalał na podłączenie do niego kamery usb udostępnionej za pośrednictwem qemu i wykonanie prostego testu - np. zrobienie zdjęcia. Jako bibliotekę do obsługi kamery użyliśmy *fswebcam*, którą załączyliśmy jako pakiet w *Target packages*. Poza tym dołączyliśmy biblioteki *libusb* oraz *libcamera*. Pełna konfiguracja systemu znajduje się w pliku *br.config* w folderze *configs*. Większej pracy wymagała od nas konfiguracja jądra. Tam z kolei dodaliśmy pakiety odnoszące się do obsługi USB i wybraliśmy dla nich odpowiednie ustawienia. Konieczne było także dołączenie pakietów V4L (Video for linux) i ustawienie sterownika dla kamery USB. Pełna konfiguracja jądra znajduje się w pliku *kernel.conf*.

7 Modyfikacje systemu

- Raspberry Pi OS lite arm64 - najlżejsza dostępna wersja, bez GUI

- Zainstalowanie Python'a i niezbędnych bibliotek wymienionych w requirements.txt oraz CMake'a
- Konfiguracja sieci WiFi i połączenia z SSH

8 Opis zawartości paczki z kodami źródłowymi

Paczka zawiera foldery:

- tracklib - zawierający algorytmy śledzenia oraz rozpoznawania gestów
- resources - zawierający logo projektu
- html - zawierający stronę odbieraną przez klienta
- configs - zawierający pliki konfiguracyjne do poprawnej konfiguracji kamery na buildroocie
- tests - zawierający testy implementacji

oraz pliki:

- main.py - zawierający program serwerowy wraz z endpointami
- requirements.txt - zawierający zależności potrzebne do uruchomienia projektu

Kod dostępny jest również pod adresem: <https://github.com/Mitchu727/SKPS—gesture-recognition>

8.1 Sposób uruchomienia

Przed uruchomieniem, należy się upewnić, że ma się zainstalowanego Python'a minimalnie w wersji 3.7.3. Podczas pierwszego uruchomienia należy pobrać wymagane pakiety znajdujące się w pliku requirements.txt, można do tego użyć komendy:

```
pip install -r requirements.txt
```

Następnie na komputerze serwerowym należy odpowiednio zmodyfikować plik index.html i umieścić w nim adres ip swojej maszyny lub adres domeny pod którym będzie dostępny serwer. Ostatecznie należy uruchomić serwer przy pomocy jednej z podanych niżej komend

```
uvicorn main:app --host 0.0.0.0 --port 8000
python3 main.py
```

Dodatkowo podczas lokalnego debugowania aplikacji przydatna może się okazać flaga -d

```
python3 main.py -d
```

Jednak do użycia tej flagi, należy mieć zainstalowaną pełną wersję biblioteki OpenCV, najlepiej w wersji 4.5.1.48

```
pip install opencv-python==4.5.1.48
```

9 Bibliografia

https://commons.wikimedia.org/wiki/File:HSV_color_solid_cylinder.png