POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka (AIR)
SPECJALNOŚĆ: Technologie Informacyjne w Automatyce (ART)

# PRACA DYPLOMOWA INŻYNIERSKA

Rozpoznawanie znaków tekstowych drogowych w systemie pomocy osobom niewidomym

The text road signs recognition in the system of helping visually-impaired people

AUTOR:
Karolina Raczyńska

PROWADZĄCY PRACĘ:
dr inż. Łukasz Jeleń

OCENA PRACY:

WROCŁAW 09.12.2016

# Spis treści

# Rozdział 1

# Wstęp

Postęp techniczny w XXI wieku sprawił, że technologie informatyczne wkroczyły już do wielu sfer naszego życia, oferując co raz to nowsze i lepsze rozwiązania. Od dziesiątek lat firmy IT skupiają się na tym, aby dotrzeć nie tylko do statystycznego klienta, ale też do tych szczególnych, ze specjalnymi potrzebami. Pojawiają się narzędzia, które pomagają funkcjonować w społeczeństwie ludziom, którzy postrzegają świat nieco innymi zmysłami - przez swoją niepełnosprawność. W świecie gdzie tysiące ludzi wpatruje się w swoje smartfony, dla ludzi niewidomych to doświadczenie jest zgoła inne. Celem producentów urządzeń mobilnych staje się rozciągnięcie swojej oferty, dla ludzi również niepełnosprawnych sensorycznie - niedowidzących czy słabosłyszących. Na rynek zaczynają napływać aplikacje, które pomagają ludziom niedowidzącym czy całkowicie niewidzącym korzystać z telefonów komórkowych. Istnieją aplikację czytające wiadomości tekstowe, nadesłane do posiadacza telefonu, a także takie które odwrotnie, zamieniają tekst mówiony na wiadomość. Pojawiają się możliwości dostosowania właściwości telefonu do potrzeb konsumenta, takie jak np. nakładki na system w telefonie, które powiększają czcionkę. Popularne jest wybieranie głosowe, które ułatwia korzystanie z telefonu nie tylko ludziom z problemami ze wzrokiem, ale też osobom starszym.

W Instytucie Nauki w Georgii grupa badaczy poczyniła wielki krok w kierunku pomocy ludziom niepełnosprawnym - stworzyła aplikację BrailleTouch, która pozwala wprowadzać tekst do smartfona, bez używania wzroku. Urządzenie musi być trzymane horyzontalnie, po obu stronach znajdują się wtenczas po trzy przyciski. Korzystając z sześciu palców, trzyma się je nad przyciskami, co nie wymaga od użytkownika przemieszczania palców po ekranie. Właściwe konfiguracje naciśnięć tych przycisków, tworzą wszystkie litery alfabetu, a także znaki specjalne. Jak nazwa aplikacji wskazuje idea ta jest oparta na alfabecie Braille'a, wystarczy dotykać odpowiednie kombinacje przycisków, a do pola tekstowego zostaje wprowadzony właściwy znak. Aplikacja jest też przygotowana na obsługę błędów. Informuje użytkownika o błędach jakie popełnił, np. przez wiadomość głosową czy wibracje [1].

Dąży się do tego aby doświadczenia sensoryczne ludzi niepełnosprawnych wizualnie, korzystających z urządzeń mobilnych były pełniejsze. W tym celu stosuje się właśnie wibracje, czy komunikaty głosowe. Właśnie te proste właściwości, wbudowane w każdym telefonie, a także ogrom algorytmów wspierających przetwarzanie obrazu, zachęcają do pisania aplikacji wspierających osoby niewidome i niedowidzące. Chce się, aby dzięki smartfonom te osoby mogły "patrzeć", korzystając jednak z innych zmysłów. Ważnym zastosowaniem telefonów w tym aspekcie jest pomoc osobom niewidomym w zachowaniu bezpieczeństwa, wykorzystanie urządzenia mobilnego jako elektronicznego przewodnika takiej osoby, ostrzegającego ją o ewentualnych zagrożeniach, które mogą ją spotkać na

drodze, np. informacja o tym, że zbliża się do strefy ruchu, gdzie może napotkać samochód, tudzież, że powinna zwrócić uwagę na roboty drogowe czy inne utrudnienia w ruchu. Rozpoznawanie otoczenia, w którym dana osoba się znajduje ma także wartość informacyjną - powiadamia, dla przykładu o nazwie ulicy na której się znajduje, jeżeli w okolicy znajduje się tabliczka z ową nazwą. Taka informacja zwrotna pod wpływem analizy otoczenia, przy wykorzystaniu urządzenia mobilnego będzie tematem niniejszej pracy inżynierskiej.

## 1.1 Cel i zakres pracy

Celem niniejszej pracy było stworzenie systemu wsparcia dla osób niewidomych i niedowidzących. Zdecydowano się zawęzić zakres pracy do napisania aplikacji mobilnej, wspierającej takie osoby. Aplikacja ta miała być na tyle intuicyjna i łatwa w obsłudze, aby osoby te mogły z niej korzystać bez większych problemów. Kolejnym celem było opisanie jej, ze szczególnym zwróceniem uwagi na algorytmy użyte przy jej powstawaniu i przeanalizowanie ich działania, a także naświetlenie pewnych informacji na temat środowiska Android. Aplikacja powinna działać na jak najmniejszej liczbie urządzeń, starając się wszystkie swoje funkcje, czyli zrobienie zdjęcie, przetworzenie go, zaprezentowanie informacji zwrotnej użytkowniki, zawrzeć w jednym urządzeniu mobilnym - telefonie z systemem operacyjnym Android, który, na dzień dzisiejszy, jest najbardziej popularnym systemem na rynku. Koniecznym warunkiem ma być jednak to, że ów telefon musi posiadać sprawny aparat i głośnik. Główną funkcjonalnością aplikacji miało być przetwarzanie zdjęć, na których znajdowały się tekstowe znaki drogowe na komunikat głosowy. Dzięki czemu poruszanie się po ulicy ludzi, mających trudność z czytaniem drogowych znaków tekstowych, stałoby się łatwiejsze, a przede wszystkim bezpieczniejsze, z uwagi na to, że tego typu znaki przekazują wiadomości istotne dla podróży niezagrażającej ludzkiemu zdrowiu, a także życiu. Pomysł niniejszej pracy powstał jako dopełnienie większego systemu wsparcia dla osób niewidomych, składającego się na rozpoznawanie tekstu mówionego i przetwarzanie elektrowibracji, stanowiącego rozwinięcie projektu zespołowego "Sztuczne Oko". Projekt ów opierał się na użyciu telefonu z aparatem, który robił zdjęcie i przetwarzał je na krawędzie, kompresował i przesyłał na Raspberry Pi, a tam konwertowane było na sygnał audio, który prowokował elektrowibracje na panelu dotykowym, ładowanym przez przetwornicę, w miejscu gdzie zarejestrowano położenie palca. Elektrowibracje te miały imitować uczucie szorstkości pod palcami, doprowadzając do możliwości "widzenia" krawędzi odczytanych ze zdjęcia, opuszkami palców.

## 1.2 Struktura pracy

Praca podzielona jest na siedem rozdziałów, pierwszy to Wstęp, wprowadzający czytelnika w temat pracy, określający jej cel i zakres. Ostatni to Zakończenie, gdzie podsumowuje się całą pracę i stwierdza czy jej cele zostały spełnione.

Drugi rozdział przedstawia podstawy teoretyczne, które zostały wykorzystane przy tworzeniu aplikacji. Przedstawiono podstawowe zagadnienia z dziedziny przetwarzania obrazu, opisano użyte algorytmy, ich działanie, zalety oraz wady, a także to jak prezentują się przy innych algorytmach rozwiązujących ten sam problem.

W trzecim rozdziale zawarto architekturę stworzonego systemu, zaprezentowano diagramy UML: klas oraz przypadków użycia. Opisano strukturę programu oraz działanie

poszczególnych klas, jakie operacje wykonywane są na obiektach tych klas, a także powiązania między nimi.

W czwartym rozdziale odniesiono się do implementacji programu. Przedstawiono środowisko w którym został napisany i zaprezentowano jego najważniejsze komponenty. Pojawiły się również przykłady kodu, wycięte z kodu źródłowego aplikacji, wraz z wytłumaczeniem ich znaczenia oraz celem wykorzystania. Opisano również zewnętrzne biblioteki, które zostały użyte.

W piątym rozdziale pokazano działanie aplikacji, dołączono screeny ekranu urządzenia mobilnego, na którym aplikacja była testowana. Dołączono opis wszystkich funkcjonalności oraz możliwych widoków.

# Rozdział 2

# Przetwarzanie obrazu

Obraz to między innymi źródło informacji - nie tylko wzrokowej. Aby uzyskać dane, które są w nim zapisane, należy go odpowiednio przetworzyć. W dziedzinie nauki, jaką jest przetwarzanie obrazów, wyróżniamy kilka etapów działania. Uproszczony schemat został przedstawiony na rysunku poniżej.


Rysunek 2.1: Schemat cyklu przetwarzania obrazu
źródło: opracowanie własne na podstawie [2]

Poniżej zostały omówione poszczególne etapy.

1. **Akwizycja obrazu** – zamiana energii świetlnej od każdego punktu sceny na sygnał elektryczny, czyli zbieranie danych o każdym pikselu. Scena jaką obserwujemy jest funkcją ciągłą, przy przetwarzaniu jej na obraz cyfrowy, pewna część danych zostaje stracona. Typowo korzysta się z dwóch sposobów pozyskiwania sygnały dyskretnego z sygnału ciągłego i jest to: kwantyzacja oraz próbkowanie. Kwantyzacja jest to pobieranie danych w funkcji wartości funkcji – w zależności od niej przyporządkowuje się próbkę do odpowiedniego przedziału, czyli kwantu, natomiast próbkowanie polega na zbieraniu danych w funkcji czasu.

2. **Przetwarzanie obrazu** – na ten etap składa się wstępna obróbka obrazu, operacje takie jak: zmniejszenie obrazu, eliminacja zakłóceń, filtracja wstępna, której celem jest wyeksponowanie na obrazie ważnych cech, takich jak krawędzie lub duże jednokolorowe obiekty. Do tego celu stosuje się np. transformację obrazu kolorowego

na skalę odcieni szarości, tudzież progowanie, w celu uzyskania obrazu binarnego, w którym piksele przyjmują jedynie jedną z dwóch wartości.

3. **Analiza obrazu** – wydobycie wcześniej wyeksponowanych cech, w celu ich późniejszego rozpoznania, już nie jako kształtu ale jako konkretnej informacji. Wynikiem tego etapu nie jest już obraz, a dane w postaci liczbowej lub statystycznej.

4. **Decyzja** – rozpoznanie obrazu, zanalizowanie cech, pozyskanie informacji, a przede wszystkim klasyfikacja [3].

Wszystkie te etapy mają miejsce w systemie wizyjnym do którego można zaliczyć np. komputer, telefon, aparat fotograficzny, a do jego podstawowych funkcji należy:

- Przyjęcie obrazu,
- zapisanie obrazu,
- właściwa obróbka obrazu,
- wyświetlenie obrazu [4];

## 2.1 Przetwarzanie do skali szarości

RGB, jest modelem przestrzeni barw, w którym każdy piksel opisują trzy wartości: R - intensywność koloru czerwonego, G - intensywność koloru zielonego oraz B - intensywność koloru niebieskiego. Obraz w odcieniach szarości to obraz w którym procentowy udział każdej z tych trzech instancji jest równy. Aby uzyskać taki wynik należy przeprowadzić pewne modyfikacje na wartościach pikseli. Przyjmuje się, że p to piksel wynikowy operacji przetwarzania do skali szarości. Najbardziej znane są trzy algorytmy:

- Desaturacja(ang. Desaturation) – metoda ta polega na tym, że z trzech wartości: intensywności koloru czerwonego, zielonego oraz niebieskiego w pikselu, zostaje wybrana wartość największa oraz najmniejsza, a wynikiem transformacji jest średnia arytmetyczna tych dwóch wartości.

$$p = \frac{max(R, G, B) + min(R, G, B)}{2} \tag{2.1}$$

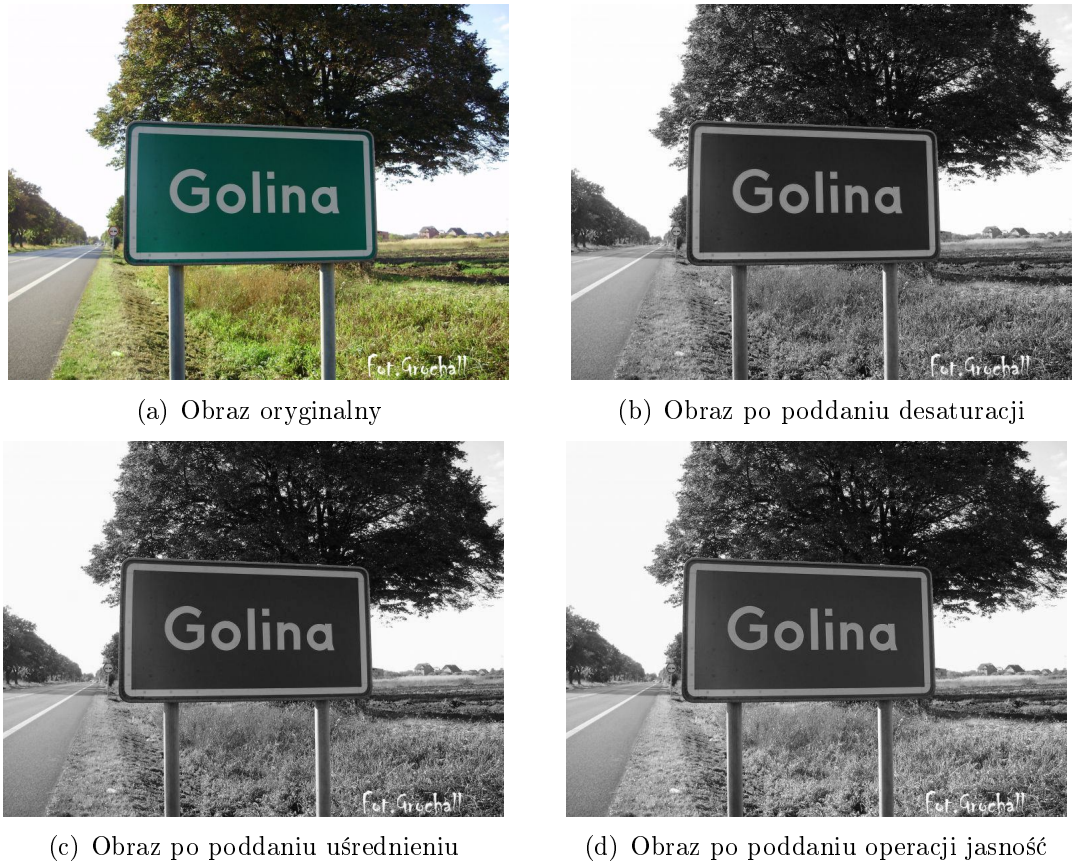
- Średnia (ang. Average) – metoda ta, najprostsza, dzieli sumę wartości intensywności trzech kolorów przez trzy.

$$p = \frac{R + G + B}{3} \tag{2.2}$$

- Jasność (ang. Luminance) – metoda również opierająca się na średniej, ale w tym wypadku jest to średnia ważona. Ludzki wzrok jest najbardziej czuły na kolor zielony, właśnie dlatego intensywność zieleni ma największą wagę. Wyjściowy piksel przyjmuje wartość [5]:

$$p = 0.21R + 0.72G + 0.07B \tag{2.3}$$

Wyniki poddania obrazu powyższym transformacją zostały przedstawione na Przykładzie 2.2.




(a) Obraz oryginalny

(b) Obraz po poddaniu desaturacji

(c) Obraz po poddaniu uśrednieniu

(d) Obraz po poddaniu operacji jasność

Przykład 2.2: Wynik przetwarzania obrazu na odcienie szarości
źródło: opracowanie własne

## 2.2 Binaryzacja

Binaryzacja to operacja punktowa. Wynikiem tej operacji jest obraz binarny, czyli taki, w którym każdy z pikseli przyjmuje tylko jedną z dwóch wartości. Ideą tego zabiegu jest wyodrębnienie obiektu od tła przez nadanie pikselom, tych dwóch instancji, różnych wartości. Wejściowym obrazem jest zazwyczaj obraz przetworzony do skali szarości, którego piksele przyjmują wartości od 0 do 255. Najczęściej stosowaną metodą klasyfikacji piksela do jednej z dwóch klas jest progowanie – sprawdzenie czy wartość piksela przekracza zadany próg czy też nie i w zależności od wyniku tego zdania logicznego – przypisanie odpowiedniej wartości. Dla progu $T$, początkowej wartości piksela $I(i,j)$ i wartości piksela po transformacji $Y(i,j)$, gdzie $i$ to współrzędna pozioma, a $j$ to współrzędna pionowa piksela na obrazie, mamy [6]:

$$I(i,j) < T \longmapsto Y(i,j) = 0 \tag{2.4}$$

$$I(i,j) \geqslant T \longmapsto Y(i,j) = 1 \tag{2.5}$$

Zasadniczą trudnością tego podejścia jest celne dobranie wartości progowej $T$. Aby wybrać odpowiednią wartość progu można zastosować jeden z poniżej zamieszczonych sposobów:

- **Na podstawie histogramu** - należy użyć histogramu poziomów szarości analizowanego obrazu. Histogram to graficzna reprezentacja rozkładu empirycznego cechy. Na wykresie zaprezentowany jest jako bloczki o odpowiedniej wysokości w zależności od liczby pikseli przyjmujących wartości z danego zakresu [7]. Następnie, w przypadku gdy histogram jest dwumodalny, to wartość progową wybiera się z pomiędzy dwóch maksimów lokalnych. Gdy ma się do czynienia z mniej szczególnym przypadkiem zastosowanie tej metody może spowodować przekłamania w wyborze progu [8].
- **Zastosowanie dwóch wartości progowych** - rozważa się progowanie z dwoma wartościami $T_1 > T_2 > 0$. Jeżeli wartości piksela znajduje się pomiędzy wartościami $T_1$ i $T_2$ przyporządkowuje mu się wartość 0, w przeciwnym wypadku 1. Tego typu podejście ma sens, kiedy tło obrazu jest o różnych poziomach szarości. Można też uśrednić poziom szarości tła. Podobnym sposobem jest zastosowanie histerezy, tutaj analizuje się jednak jeszcze sąsiednie piksele [9].
- **Automatyczny dobór wartości progowej** - próg wybierany jest programowo, korzystając z pewnego algorytmu. Te metody bazują na pewnych założeniach, histogram musi być bimodadalny, obiekty muszą być w kolorach przeciwnych do kolorów tła. Do metod automatycznych zalicza się przedstawioną szerzej metodę Otsu [10].

### 2.2.1 Metoda Otsu

Metoda Otsu należy do rodziny metod optymalnych względem pewnej funkcji kryterialnej. W tym przypadku ową funkcja kryterialną jest wariancja wewnątrzklasowa oraz międzyklasowa. Metoda ta jest bardzo prosta, w uogólnieniu używa momentów zerowego oraz pierwszego stopnia histogramu obrazu.

Pierwszym etapem poszukiwania wartości progowej metodą Otsu jest normalizacja histogramu obrazu w skali szarości. Zakłada się, że piksele przyjmują wartości z $L$ poziomów, a do każdego i-tego poziomu jest przydzielana $n_i$ liczba pikseli. Przyjmując, że $N$ to liczba wszystkich pikseli obrazu, rozkład prawdopodobieństw $p_i$ tego histogramu przedstawiony jest następująco:

$$p_i = \frac{n_i}{N} \qquad p_i \geqslant 0, \sum_{i=1}^{L} p_i = 1 \tag{2.6}$$

Następnym krokiem jest dychotomia pikseli, czyli podział wszystkich elementów na dwie klasy względem pewnego progu $t$. Wtenczas piksele, które przyjmują wartości mniejsze od zadanej wartości progowej $t$ zostaną sklasyfikowane do klasy $K_0$, a te większe do klasy $K_1$. Wtedy prawdopodobieństwo $\omega_0$ oraz $\omega_1$ tego, że badany piksel zostanie przypisany do danej klasy jest określone wzorami:

$$\omega_0 = Pr(K_0) = \sum_{i=1}^{t} p_i = \omega(t) \tag{2.7}$$

$$\omega_1 = Pr(K_1) = \sum_{i=t+1}^{L} p_i = 1 - \omega(t) \tag{2.8}$$

Średnia wartość piksela $mu_0$ oraz $mu_1$ przyjmowana w klasie określona została w poniższych wzorach:

$$\mu_0 = \sum_{i=1}^{t} Pr(i|K_0) = \sum_{i=1}^{t} \frac{ip_i}{\omega_0} = \frac{\mu(t)}{\omega(t)} \tag{2.9}$$

$$\mu_1 = \sum_{i=t+1}^{L} iPr(i|K_1) = \sum_{i=t+1}^{L} \frac{ip_i}{\omega_1} = \frac{\mu_T - \mu(t)}{1 - \omega(t)} \tag{2.10}$$

Gdzie:

$$\omega(t) = \sum_{i=1}^{t} p_i \tag{2.11}$$

$$\mu(t) = \sum_{i=1}^{t} ip_i \tag{2.12}$$

Powyższe wartości są skumulowanymi momentami zerowego oraz pierwszego stopnia obrazu dla wartości progowej równej $t$. A średnia z wartości pikseli na całym obrazie może zostać zapisana wzorem:

$$\mu_T = \mu(L) = \sum_{i=1}^{L} ip_i \tag{2.13}$$

Niezależnie od wyboru wartości parametru $t$, prawdziwe są równania:

$$\omega_0\mu_0 + \omega_1\mu_1 = \mu_T, \qquad \omega_0 + \omega_1 = 1. \tag{2.14}$$

Wariancje obu klas można przedstawić następująco:

$$\sigma_0^2 = \sum_{i=1}^{t} (i - \mu_0)^2 Pr(i|K_0) = \sum_{i=1}^{t} (i - \mu_0)^2 \frac{p_i}{\omega_0} \tag{2.15}$$

$$\sigma_0^2 = \sum_{i=t+1}^{t} (i - \mu_1)^2 Pr(i|K_1) = \sum_{i=t+1}^{L} (i - \mu_1)^2 \frac{p_i}{\omega_1} \tag{2.16}$$

W kolejnym etapie należy sprawdzić czy wartość zmiennej $t$ jest wartością optymalną względem funkcji kryterialnej. Funkcją kryterialną, jak wcześniej wspomniano jest:

- Wariancja wewnątrzklasowa, dąży do minimalizacji,

$$\lambda = \frac{\sigma_B^2}{\sigma_W^2}, \quad \sigma_W^2 = \omega_0\sigma_0^2 + \omega_1\sigma_1^2, \quad \sigma_B^2 = \omega_0\omega_1(\mu_1 - \mu_2)^2 \tag{2.17}$$

- Wariancja międzyklasowa, dąży do maksymalizacji.

$$\kappa = \frac{\sigma_T^2}{\sigma_W^2}, \quad \sigma_T^2 = \sum_{i=1}^{L} (i - \mu_T)^2 p_i \tag{2.18}$$

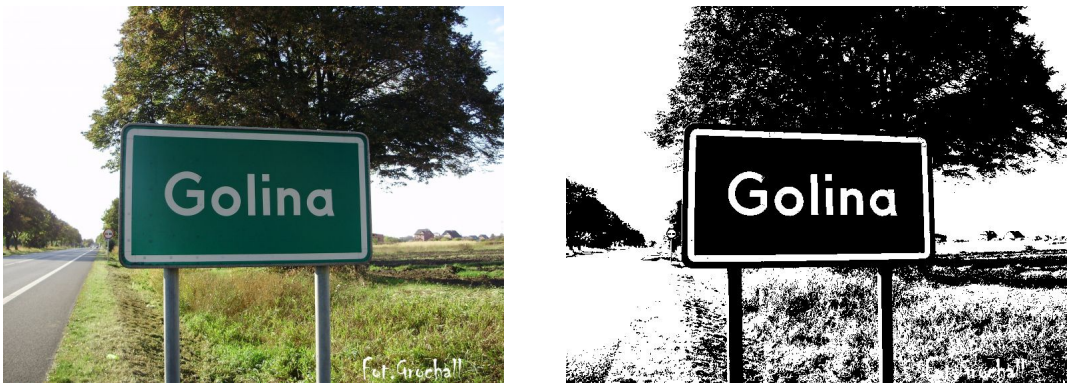
Teoria ta intuicyjnie zgadza się z pożądanym efektem działania algorytmu. Chce się aby wartości pikseli w klasach były do siebie jak najbardziej zbliżone, a między klasami występowała jak największa różnica. Kryterium jakie stosuje się do optymalizacji wartości $t$ jest łączna wariancja poziomów $\eta$. Parametr jest optymalny, jeżeli $\eta$ przyjmuje wartość jak największą, tym samym wariancja wewnątrzklasowa. Wartość ta będzie zerowa jedynie w przypadku gdy wszystkie piksele obrazu zostaną zakwalifikowane tylko do jednej z klas, jednak jest to sprzeczne z celem działania algorytmu, więc w tym kontekście maksimum zawsze istnieje.

$$\eta = \frac{\sigma_B^2}{\sigma_T^2}, \tag{2.19}$$

Zaletą tej metody jest jej prostota – korzysta się jedynie z momentów zerowego i pierwszego rzędu, a także stabilność – optymalny próg jest wybrany automatycznie, nie

bazuje na zróżnicowaniach (takich jak lokalne zbiory pikseli o niskich lub wysokich wartościach), skupia się na globalnych właściwościach obrazu, histogramie. Ważną cechą jest też uniwersalność metody, dzięki, której można ją wykorzystywać w różnych dziedzinach analizy obrazu. Może służyć do binaryzacji, jak wykorzystano w projekcie, wykorzystywana jest też przy progowaniu [11]. Wadą tej metody jest możliwość błędu poprzez wybranie niewłaściwej wartości początkowej $t$.

Wynik poddania obrazu powyższej transformacji został przedstawiony na Przykładzie 2.3.




(a) Obraz oryginalny (b) Obraz po progowaniu

Przykład 2.3: Wynik progowania obrazu metodą Otsu
źródło: opracowanie własne

## 2.3 Wykrywanie krawędzi

Przetwarzanie obrazów opiera się na wydobywaniu z obrazu takich cech, które są istotne przy późniejszej analizie i identyfikacji obiektów na nim zawartych. Jedną z takich cech jest krawędź, czyli znaczna lokalna zmiana w intensywności obrazu, związana z brakiem ciągłości w intensywności obrazu lub w pierwszej pochodnej intensywności obrazu. Występują dwa rodzaje takiej zmiany ciągłości:

1. **Nieciągłość skokowa** - intensywność obrazu zmienia się z jednej wartości w drugą i przez jakiś czas tą wartość utrzymuje, przypomina odpowiedź skokową, na obrazie może być to moment styku dwóch obiektów;

2. **Nieciągłość liniowa** - intensywność obrazu zmienia się z jednej wartości w drugą lecz w krótkim okresie czasu do niej powraca, przypomina odpowiedź impulsową, na obrazie może to być przerwanie w obiekcie;

W praktyce jednak tego typu krawędzie zdarzają się dość rzadko, krawędzie nie są aż tak wyraziste i skok przypomina bardziej rampę, a linia - dach [12].

Wykrywanie krawędzi to przede wszystkim rozpoznawanie znacznych zmian w obrazie, jest to mocno związane z wyznaczaniem maksimum pierwszej pochodnej.

Gradient jest miarą zmiany, wektorem, który wskazuje kierunki i szybkość wzrostu wartości, to dwuwymiarowy odpowiednik pierwszej pochodnej. W obrazie zmianą jest różnica w intensywności koloru. To właśnie on jest bazą większości algorytmów wykrywających krawędzie. Do operatorów gradientowych należy: operator Robertsa, gdzie krawędź to tutaj maksimum pierwszej pochodnej gradientu, operator Prewitta, który opiera się na

wygładzeniu obrazu w kierunku ortogonalnym do kierunku w którym wyznaczana jest pochodna oraz operator Sobela, gdzie krawędź jest również wyznaczana przez maksimum pierwszej pochodnej i używany jest jednowymiarowy filtr uśredniający. Gradient obrazu dany jest wzorem:

$$\nabla f = \begin{pmatrix} g_x \\ g_y \end{pmatrix} = \begin{pmatrix} \frac{\delta f}{\delta x} \\ \frac{\delta f}{\delta y} \end{pmatrix} \tag{2.20}$$

Gdzie:
$g_x$- gradient w kierunku x,
$g_y$- gradient w kierunku y
Kierunek zmian można wyliczyć z poniższego wzoru:

$$\theta = \tan^{-1} \begin{pmatrix} g_x \\ g_y \end{pmatrix} \tag{2.21}$$

Etapy wykrywania krawędzi:

1. **Filtracja**

   Aby ulepszyć działanie detektora krawędzi należy pozbyć się szumów, które utrudniają wyliczanie gradientów. Należy jednak zachować umiar, zbyt duże wygładzanie obrazu może spowodować osłabienie krawędzi. Często w tym celu stosuje się rozmycie Gaussowskie, które zostało omówione szerzej w podrozdziale 2.3.1.

2. **Wzmocnienie**

   W polach obrazu gdzie umiejscowione są największe zmiany intensywności koloru podkreśla się te piksele, które mają na tą zmianę największy wpływ. Używa się w tym celu gradientu.

3. **Wykrycie**

   W związku z licznymi zanieczyszczeniami i szumami obecnymi na obrazie zdarza się, że piksele o niezerowym gradiencie nie należą do krawędzi. Aby pominąć piksele nieistotne, wybiera się próg wedle którego ocenia się czy piksel należy do krawędzi czy też nie.

4. **Zlokalizowanie**

   Etap ten składa się na większość algorytmów, polega na określeniu rozdzielczości, położenia pikseli, które składają się na krawędź. Jest to zdecydowanie przydatne w momencie potrzeby wyodrębnienia z obrazu konkretnego obiektu [13].

### 2.3.1 Detektor Canny

Detektor ten jest najczęściej używanym detektorem w świecie przetwarzania obrazów. Należy do rodziny detektorów Gaussowskich, specjalizujących się w wykrywaniu krawędzi skokowych. Jest to algorytm optymalny względem kilku kryteriów:

- Poprawna detekcja - dąży się do minimalizacji klasyfikacji nieistniejących krawędzi jako krawędzi istniejących. Są większe szanse na pominięcie krawędzi niż na wykrycie krawędzi błędnej.
- Poprawna lokalizacja - dąży się do tego, aby wykryte krawędzie znajdowały się jak najbliżej istniejących krawędzi.

- Jednoznaczna odpowiedź - dąży się do minimalizacji lokalnych maksimów wokół wykrytej krawędzi, idea jest taka, aby dla każdego istniejącego punktu krawędzi zwracany był tylko jeden piksel.

Sposób działania algorytmu można opisać w kilku krokach:

Krok 1 Obraz wejściowy należy poddać operacji rozmycia Gaussa, w celu pozbycia się szumów. Polega to na nałożeniu na obraz maski, która jest dyskretną aproksymacją funkcji Gaussa opisaną wzorem poniżej, gdzie $\sigma$ jest stałą:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.22}$$

Matematycznie, nałożenie maski na obraz to wyliczenie splotu dyskretnych wartości funkcji Gaussa i pikseli obrazu [14]. Rozmycie Gaussa jest zależne od wartości gradientu.

Krok 2 Następnie wyliczona zostaje wartość oraz kierunek gradientu, korzystając odpowiednio ze wzorów 2.21 oraz 2.22.

Krok 3 Wykrywa się punkty należące do krawędzi i tłumi te wartości, które nie są maksimami lokalnymi, aby pozbyć się niewyraźnych krawędzi.

Krok 4 W ostatnim etapie punkty krawędzi zostają połączone i wybiera się histerezę - przedział od najniższej wartości progu do najwyżej wartości progu w jakim ma się znaleźć krawędź. Najwyższy próg $k_{max}$ dotyczy mocnych krawędzi, natomiast najniższy próg $k_{min}$ dotyczy krawędzi słabych.

Wynik poddania obrazu powyższej transformacji znajduje się poniżej na Przykładzie 2.4.

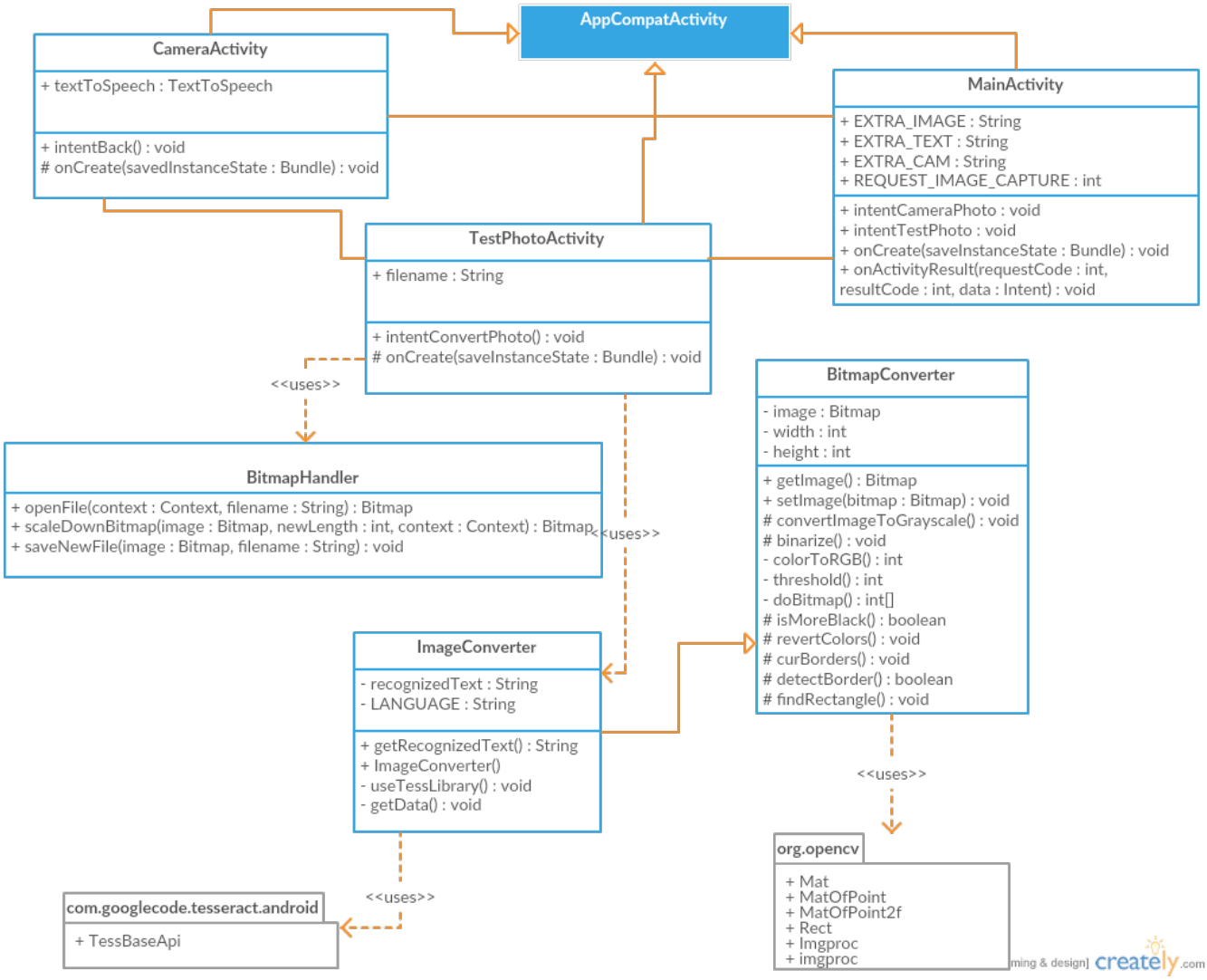

(a) Obraz oryginalny (b) Obraz po wykryciu krawędzi

Przykład 2.4: Wynik wykrywania krawędzi detektorem Canny
źródło: opracowanie własne

# Rozdział 3

# Architektura systemu

Program został napisany, korzystając z obiektowości języka Java. Wykorzystano klasy oraz aktywności, które wzajemnie ze sobą powiązano, w celu minimalizacji ilości kodu, a także elastyczności kodu na zmiany. Diagram klas został przedstawiony na Rysunku 3.1. Wykonany został darmowym narzędziem *creatly*.




Rysunek 3.1: Diagram UML klas
źródło: opracowanie własne

Komunikacja między Aktywnościami została zaimplementowana, korzystając z intencji, pojęcia te zostały wyjaśnione w paragrafie 4.1.2. Poniżej zostanie zcharakteryzowana każda z klas i Aktywności:

- **AppCompatActivity** to klasa bazowa, używana w przypadku Aktywności, które korzystają z dodatkowych bibliotek. Wchodzi w skład podstawowych bibliotek systemu Android [15].

- **MainActivity** to Aktywność, która dziedziczy po **AppCompatActivity**. Obsługuje Widok główny aplikacji, zaprezentowany na rysunku 5.1. Składa się z dwóch obiektów typu *Button*, które przekierowują do dwóch różnych Widoków. Jeden jest obsługiwany przez kolejną Aktywność - **TestPhotoActivity** i zostanie opisany później, a drugi - kamera urządzenia mobilnego, jest również obsługiwany przez **MainActivity**. W Intencji, zrobione zdjęcie przesyłane jest do funkcji *onActivityResult*, gdzie zostaje odpowiednio przetworzone, tekst zostaje rozpoznany i w kolejnej Intencji wyniki tych dwóch operacji zostają przesłane do **CameraActivity**.

- **TestPhotoActivity** to Aktywność, która również dziedziczy po **AppCompatActivity**. Zajmuje się obsługą Widoku testowego, wyświetla obiekty typu *ImageButton*, które po kliknięciu przesyłają Intencję z odpowiednim obrazem, który ma zostać poddany przetwarzaniu. Widok ten jest przedstawiony na rysunku 5.2. Przetworzony obraz wraz z rozpoznanym tekstem zostają przesłane do **CameraActivity**.

- **CameraActivity** to Aktywność, która także dziedziczy po **AppCompatActivity**. Zajmuje się obsługą Widoku, który reprezentuje końcowy wynik działania aplikacji. Widok ten można znaleźć na rysunkach 5.3b, 5.4b oraz 5.5b. Sam Widok zawiera obiekty: *ImageView*, służący do wyświetlenia przetworzonego obrazu, *TextView*, służący do wyświetlenia odczytanego tekstu, oraz dwa obiektu typu *Button*, gdzie jeden przekierowuje użytkownika do poprzedniego Widoku, a drugi pozwala na powtórzenie komunikatu głosowego, który jest odczytywany z przekazanego do aktywności tekstu. W klasie odczytywana jest Intencja przesłana z Aktywności **MainActivity**, pobierany jest tekst, który rozpoznano na obrazie, typu String, następnie tekst ten zostaje odczytany, korzystając z klasy *TextToSpeech*, która została szerzej opisana w paragrafie 4.1.1.

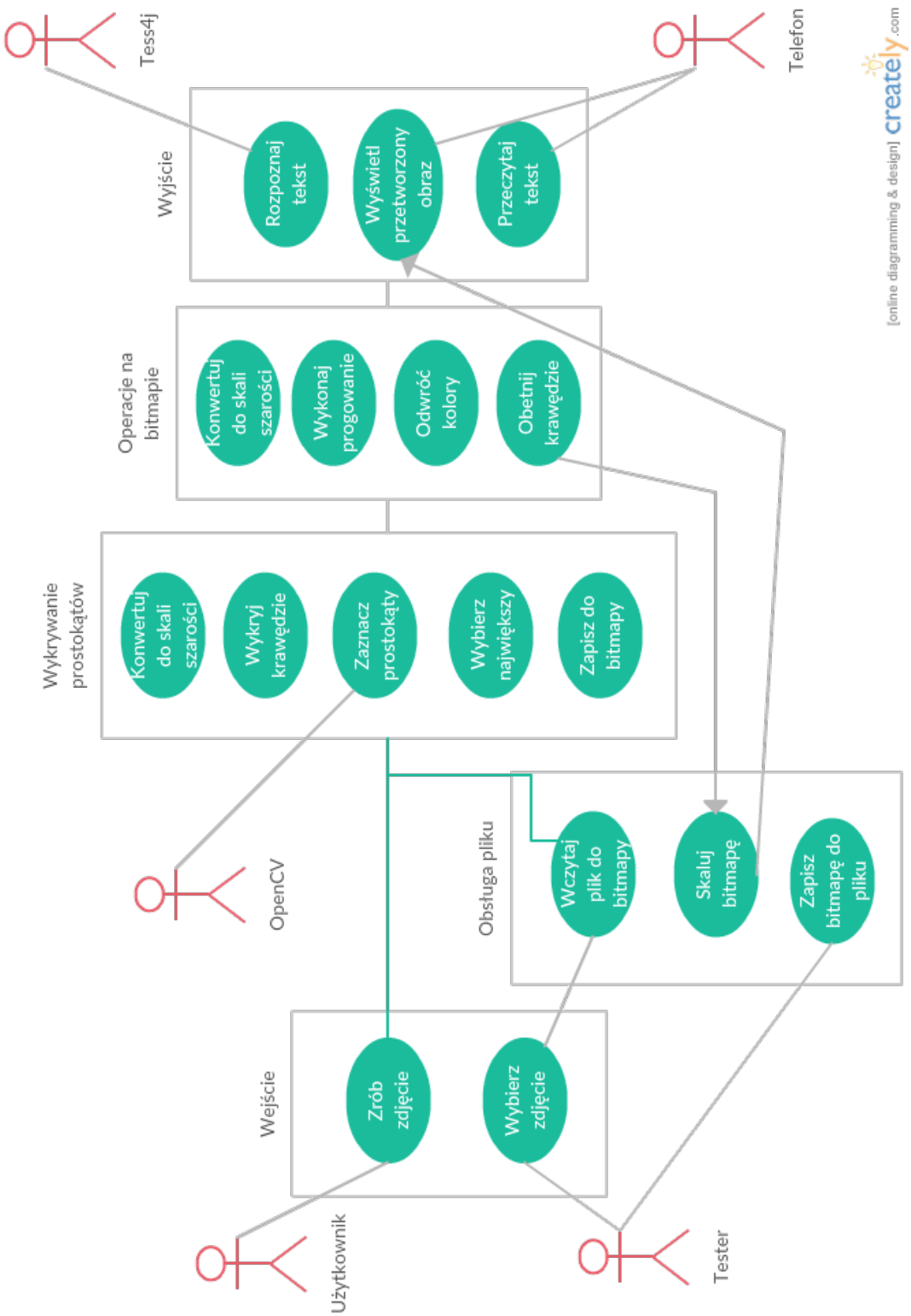
- **BitmapHandler** to klasa, która zajmuje się obsługą plików i bitmap. Konwertuje obraz w pliku do obiektu typu Bitmap w funkcji *openFile*. Zapisuje bitmapę do pliku w funkcji *saveNewFile*. Daje też możliwość przeskalowania bitmapy do mniejszego rozmiaru w funkcji *scaleDownBitmap*, co jest przydatną rzeczą, o tyle, o ile Android jest systemem, który jest w stanie sobie poradzić z obróbką większych obrazów, to już aby je wyświetlić, rozmiarowo należy dopasować je do ekranu urządzenia mobilnego na którym działa aplikacja.

- **BitmapConverter** to klasa, która zajmuje się zasadniczą trudnością niniejszego projektu - przetwarza bitmapę w taki sposób, aby biblioteka Tess4j poradziła sobie z rozpoznaniem na niej tekstu. Wykorzystuje w tym celu bibliotekę OpenCV, a także własne, zaimplementowane funkcje. Na początek wycina ze zdjęcia prostokąt, przyjmując, że jest on znakiem drogowym, korzystając z funkcji $findRectangle$, której działanie zostało opisane w paragrafie 4.3.1. Kolejno prostokąt zostaje poddany operacjom:

1. Przetworzenia do skali szarości w funkcji *convertImageToGrayscale*, korzystając z metody Jasność, opisanej w paragrafie 2.1.
2. Progowania metodą Otsu w funkcji *binarizeImage*, metoda ta została opisana w paragrafie 2.2.1.
3. Sprawdzenia czy jest więcej koloru czarnego na obrazie w funkcji *isMoreBlack* i odwrócenia kolorów w funkcji *revertColors*, gdy czarny kolor występuje częściej, korzystając z założenia, że jeżeli chce się otrzymać czarny tekst na białym tle, to procentowo białe tło składa się z większej liczby pikseli. Problem, wraz z rozwiązaniem, szerzej opisany w paragrafie 4.4.
4. Sprawdzenia czy obraz jest obramowany w funkcji *detectBorders*. I jeżeli jest to pozbycie się tych obramowań w funkcji *cutBorders*. Obie funkcje szerzej opisane w paragrafie 4.4.

- **ImageConverter** to klasa, która dziedziczy po **BitmapConverter** i korzysta z biblioteki Tess4j. Konstruktor tej klasy wykonuje całe przetwarzanie obrazu i rozpoznanie tekstu. Przyjmuje zdjęcie w formacie Bitmap, wykonuje wszystkie operacje opisane we wcześniejszym punkcie, następnie przetwarza wynikową bitmapę na tekst i ten tekst jest wartością zwracaną konstruktora. Klasa ta powstała w celu rozdzielenia metod przetwarzających bitmapę i tych związanych z rozpoznawaniem tekstu. Zmniejszyła także liczbę komend wywoływanych w Intencjach, przesyłanych do Widoku z wynikiem działania aplikacji. W Intencjach inicjalizuje się jedynie obiekt klasy, korzystając z konstruktora. Obiekt ten składa się z przetworzonego obrazu i rozpoznanego tekstu.

Działanie programu można podzielić na kilka etapów, jak na Rysunku 3.2, przedstawiającym diagram przypadków użycia. Został on wykonany, korzystając z aplikacji *creatly*. Mamy: wejście, obsługę pliku, wykrywanie prostokątów, operacje na bitmapie oraz wyjście. Zaczynając od wejścia, pojawiają się dwie opcje w zależności od typu aktora, użytkownik będzie chciał zrobić zdjęcie obiektowi znajdującym się przed nim, więc wybierze opcję "Zrób zdjęcie", natomiast tester będzie chciał przetestować działanie aplikacji na znanych mu danych testowych, dlatego wybierze opcję "Wybierz zdjęcie". Kolejno wybrane zdjęcie będzie musiało zostać pobrane z pliku, następnie program zajmie się wykrywaniem prostokątów, zdjęcie z kamery od razu trafia do tego bloczka. Tutaj uczestniczy aktor, którym jest biblioteka OpenCV. Następnie prostokąty są odpowiednio przetwarzane wcześniej opisanymi operacjami na bitmapie. Dalej rozpoznaje się tekst, w czym uczestniczy aktor: Tess4j, i na urządzeniu mobilnym wyświetla się przetworzony obraz, a tekst zostaje przeczytany.




Rysunek 3.2: Diagram UML przypadków użycia
źródło: opracowanie własne

# Rozdział 4

# Implementacja

Projekt został napisany pod system operacyjny Android, a stworzony na platformie Android Studio, w związku z czym językiem implementacji była Java. W niniejszej pracy wykorzystano dwie otwarte, darmowe biblioteki dla platformy Java: Tess4j oraz OpenCV. Tess4j - bibliteka OCR, przetwarzała obraz na mowę, a z biblioteki OpenCV, która oferuje bardzo dużo funkcji do przetwarzania obrazów wykorzystano funkcję wykrywającą na obrazie prostokąty.

## 4.1 System operacyjny Android

Aplikacja została napisana pod system operacyjny Android, jako, że ma on największy procentowy skład na rynku telefonów - 85% [16]. Natywnym jeżykiem w jakim pisze się aplikacje na system operacyjny Android jest Java. Jednak istnieją platformy, które pozwalają pisać również w innych językach np. C#, korzystając z Xamarin. Xamarin był wcześniej platformą niezależną, ale po wykupieniu przez firmę Microsoft został dodany jako nakładka do Visual Studio 2015 w Aktualizacji 2. Języki C# i Java są do siebie bardzo podobne, działają nawet na ekwiwalentnych wirtualnych maszynach - Java na JVM(Java Virtual Machine), a C# na .NET. Jeszcze kilka lat temu największym atutem Javy był fakt, że jest ona rozwiązaniem międzysystemowym, działającym zarówno na systemie Windows jak i Linux [17], jednak aktualne aktualizacje .NETowe sprawiły, że różnica ta się rozmyła. Jednakże w tej pracy zdecydowano się na język natywny aplikacji androidowych, czyli Javę, z uwagi na o wiele bogatszą dokumentację, zarówno w języku polskim i angielskim, a także znaczną liczbę kursów i artykułów pomocnych przy pisaniu kodu. Java jest językiem, który radzi sobie z tymi przeszkodami, z którymi nie radzą sobie języki skryptowe i jest jednym z najpopularniejszych takich języków. Radzi sobie z takimi problemami jak dostęp do baz danych, przetwarzanie rozproszone, programowanie wielowątkowe, a także sieciowe [18]. Jako IDE wykorzystano oficjalne dla platformy - Android Studio. Pierwsze implementacje były pisane na platformie Intellij IDEA, aby ułatwić debuggowanie oraz testowanie kodu, które trwało tutaj zdecydowanie krócej niż na podłączonym urządzeniu tudzież wirtualnej maszynie. Android Studio jest produktem stworzonym przez tą samą firmę, w związku z czym migracja między tymi dwiema platformami była najrozsądniejszym rozwiązaniem.

### 4.1.1 TextToSpeech - zamiana tekstu na mowę

Do przeczytania tekstu - konwersji obiektu typu String na mowę - użyto klasy TextToSpeech, znajdującej się w bibliotece platformy Android. Instancja tej klasy, po odpo-

wiedniej inicjalizacji, od razu odtwarza dźwięk, który czyta podany tekst. Odpowiednia inicjalizacja polega na zaimplementowaniu *TextToSpeech.OnInitListener*[19]. Inicjalizacja instancji następuje wtedy kiedy zostaje ona wywołana. Ustawia się tam język w jakim użytkownik chce aby tekst został przeczytany. Wartość języka przechowuje się w zmiennej *locale*. W przypadku ustawienia wartości domyślnej, tekst powinien zostać przeczytany w języku jaki ustawiony jest na urządzeniu mobilnym na którym działa aplikacja. Do zmiennej *toSpeak* zostaje przesłana wartość z Intencji - wynik działania Tess4j.

```
TextToSpeech textToSpeech=new TextToSpeech(getApplicationContext(),
new TextToSpeech.OnInitListener() {
    @Override
    public void onInit(int status) {
        if(status != TextToSpeech.ERROR) {
            Locale locale = new Locale("pl","PL");
            textToSpeech.setLanguage(locale);
        }
     }
});

final String toSpeak = getIntent().getExtras().getString(MainActivity.
   EXTRA_TEXT);
textToSpeech.speak(toSpeak, TextToSpeech.QUEUE_FLUSH, null);
```

Kod źródłowy 4.1: Kod programu, który odpowiada za przeczytanie tekstu z wykorzystaniem klasy TextToSpeech

### 4.1.2 Intencje i Aktywności - komponenty platformy Android

**Aktywność**(ang. *Activity*) to pojedyncza, skondensowana funkcjonalność, jaką może wykonać użytkownik. Skoro praktycznie każda Aktywność jest związana z interakcją z użytkownikiem, więc klasa ta zajmuje się tworzeniem **Widoków**(ang. *View*) z interfejsem użytkownika, które ustawiane są za pomocą funkcji *setContenView*(*View*), gdzie zmienna *View* to Widok, który programista chce połączyć z daną Aktywnością [20]. Widok to klasa reprezentująca GUI, czyli graficzny interfejs użytkownika. Korzysta z interaktywnych komponentów jak *Button*, czyli obiekt, reagujący w momencie kiedy się go naciśnie, wykonując akcję opisaną w Aktywności połączonej z tym Widokiem [21]. Widok może być określony w pliku XML lub zaprogramowany metodą *draganddrop*(z ang. *weiupu*) - pojedyncze komponenty są pobierane z biblioteki oferowanej przez środowisko Android Studio, i umiejscawiane w odpowiednich regionach na prostokątnym polu, imitującym ekran telefonu. Cechy owych komponentów, takie jak nazwa, położenie, zależności między pozostałymi komponentami i inne są również ustawiane, poprzez wybieranie wartości cech z listy lub wpisywanie ich do odpowiednich pól.

**Intencja**(ang. *Intent*) to kolejny komponent androidowej aplikacji, abstrakcyjny opis operacji, która ma zostać wykonana [22]. Umożliwia komunikację między aplikacjami, a także wewnątrz pojedynczej aplikacji - pomiędzy Aktywnościami. Intencje, przez pryzmat ich drugiej własności - komunikacji między Aktywnościami, wykorzystanej w tej pracy, można podzielić na dwa rodzaje [23]:

- **Jawne** - w konstruktorze Intencji, obiekt, który ma wykonać zadanie jest ściśle określony - nazwą jego klasy.

- **Niejawne** - zadeklarowana jest konkretna akcja, ale nie jest powiedziane jaki komponent ma ją wywołać[24].

Poniżej zaprezentowano działanie intencji, wykorzystane w niniejszym projekcie. W Kodzie źródłowym 4.2 nastąpiło jawne zadeklarowanie Intencji w Aktywności - **MainActivity** i będzie ona wykonana w Aktywności **CameraActivity**. W Kodzie źródłowym 4.3 do Intencji zostają dodane obiekty różnego typu - przetworzony obraz oraz rozpoznany tekst. W Kodzie źródłowym 4.4 Intencja zostaje wywołana. Następnie w Kodzie źródłowym 4.4 pokazano jak obsłużono Intencje w klasie **CameraActivity**. Funkcją *getIntent* pobrano Intencję, a następnie wszystkie dodatkowe obiekty, które zostały do niej dołączone, odwołując się do klasy z której pochodzą zmienną typu String.

```
Intent intent = new Intent(this, CameraActivity.class);
```

Kod źródłowy 4.2: Zadeklarowanie intencji

```
extra.putParcelable(MainActivity.EXTRA_IMAGE, imageConverter.getImage());
extra.putString(MainActivity.EXTRA_TEXT, imageConverter.getRecognizedText()
    );
intent.putExtras(extra);
```

Kod źródłowy 4.3: Przekazanie obiektów różnego typu w intencji

```
startActivity(intent);
```

Kod źródłowy 4.4: Wywołanie intencji

```
imageView.setImageBitmap((Bitmap) getIntent().getExtras().getParcelable(
    MainActivity.EXTRA_IMAGE));
final String toSpeak = getIntent().getExtras().getString(MainActivity.
    EXTRA_TEXT);
```

Kod źródłowy 4.5: Obsługa intencji w klasie w którym ma zostać wykonana

## 4.2 OCR - optyczne rozpoznawanie znaków

OCR (ang. *OpticalCharacterRecognition*) to system przetwarzania dokumentów w formie papierowej na tekst, w taki sposób, aby otrzymanym wynikiem były cyfrowe informacje gotowe do dalszego przetwarzania. Ten zbiór technik jest często wykorzystywany w biznesie, przechowywanie wszystkich dokumentów zajmuje wiele miejsca, po zamianie ich na postać cyfrową - miejsce jest zaoszczędzane, a dane gotowe do ewentualnej dalszej obróbki [25]. OCR wykorzystywany jest także przy odczytywaniu tablic rejestracyjnych samochodów, w technikach stosowanych przez policję [26]. W niniejszej pracy techniki OCR zostały zastosowane do odczytania z odpowiednio przetworzonego obrazu - tekstu, i zapisanie go w obiekcie typu String.

### 4.2.1 Biblioteka Tess4j

Tess4j to opensource'owa biblioteka. Jest to nakładka na Tesseract OCR API dla platformy Java. Język w jakim napisana jest biblioteka Tesseract to C++. Biblioteka ta umożliwia optyczne rozpoznawanie znaków (OCR) umieszczonych w plikach w formatach

- TIFF, JPEG, GIR, PNG oraz BMP, wielostronicowe obrazy TIFF, a także dokumenty w formacie PDF [27]. Poniżej została przedstawiona część kodu programu. Zostaje stworzony obiekt klasy *TessBaseAPI*, określa się miejsce w urządzeniu mobilnym, gdzie znajduje się folder *tessdata* - z danymi uczącymi w języku polskim, jeżeli na urządzeniu ów folder się nie znajduje, zostaje on ściągnięty z repozytorium umieszczonym w sieci na platformie Github. Następnie rozpoczyna się działanie aplikacji funkcją *init*, gdzie podaje się ścieżkę do pliku z danymi uczącymi oraz język w jakim mają być rozpoznawane słowa. Kolejno do funkcji *setImage* podaje się obraz w jednym ze wcześniej przytoczonych formatów. Obraz ten musi być odpowiednio przetworzony, gdyż biblioteka nie wykrywa miejsca położenia tekstu na zdjęciu:

- Tekst nie może być pisany ręcznie.
- Tekst musi być w kolorze czarnym.
- Tekst musi być umiejscowiony na białym tle.
- Na obrazie może znajdować się tylko tekst.

```
TessBaseAPI baseApi = new TessBaseAPI();
baseApi.setDebug(true);
File tessdataFolder = new File(
Environment.getExternalStorageDirectory().getAbsoluteFile() + "/tessdata");
if (!tessdataFolder.exists()) {
    getData();
}
String path = String.valueOf(
Environment.getExternalStorageDirectory()) + "/";
baseApi.init(path, LANGUAGE);
baseApi.setImage(bitmap);
String recognizedText = baseApi.getUTF8Text();
baseApi.end();
```

Kod źródłowy 4.6: Kod programu, odpowiadający za odwołanie do zewnętrznego API Tess4j

## 4.3 Biblioteka OpenCV

Biblioteka OpenCV to zbiór funkcji wykorzystywanych przy przetwarzaniu obrazów, jest otwarta i darmowa oraz można z niej korzystać na różnych systemach, jak Mac OS X, Windows i Linux. Biblioteka ta została napisana w jeżyku C, ale korzystając z odpowiednich nakładek można ją stosować również w języku takim jak Java.

Obraz w bibliotece OpenCV jest traktowany jako macierz, przecięcia kolumn i wierszy opisują położenie piksela i zawierają jego wartość, albo w postaci jednej liczby gdy obraz jest w skali szarości, albo w postaci trzech liczb, gdy obraz jest kolorowy. Wystpuje kilka typów zapisu danych Arr, Scalar i Mat. Ten ostatni został wykorzystany w niniejszej pracy [28].

### 4.3.1 Wykrywanie prostokątów

W celu użycia metody *boundingRect*, która wykrywa prostokąty na obrazie należało odpowiednio przygotować obraz wejściowy. Należało przetworzyć go do odcieni szarości,

a następnie wyrysować jego krawędzie, metodą Canny. W poniższym kodzie tak przetworzony obraz znajduje się w zmiennej typu Bitmap o nazwie *converted*. Ze wszystkich wykrytych krawędzi, które tworzą obiekty przeanalizowano wszystkie te, które mają kształt prostokąta. Za każdym razem liczono pole wykrytej figury i sprawdzano czy jest większa od poprzedniego prostokąta. Korzystano z założenia, że znaki drogowe tekstowe mają kształt prostokątów i na zdjęciu zcentrowanym na taki znak, założono, że będzie on największym prostokątem. Następnie tworzy się nowy obiekt typu Bitmap o wymiarach znalezionego prostokąta i w wyniku działania poniższej funkcji otrzymuje się obiekt zawierający jedynie prostokąt, czyli w domniemaniu znak drogowy, gotowy do dalszej obróbki. Prostokąt ten jest wycięty z oryginalnego zdjęcia, z uwagi na to, że skorzystanie z detektora Canny o dużej rozpiętości progów zarysowało najważniejsze krawędzie, ale mogło zignorować napisy.

```
List<MatOfPoint> contours = new ArrayList<MatOfPoint>();
Imgproc.findContours( converted, contours, new Mat(), Imgproc.RETR_LIST,
    Imgproc.CHAIN_APPROX_SIMPLE);
MatOfPoint2f approximateCurve = new MatOfPoint2f();
Bitmap bitmap = null;
int max = 0;
Mat copyMat = new Mat();
   for (int i=0; i<contours.size(); i++) {
       MatOfPoint2f contour2f = new MatOfPoint2f( contours.get(i).toArray()
            );
       double approximateDistance = Imgproc.arcLength(contour2f, true)
          *0.02;
       Imgproc.approxPolyDP(contour2f, approximateCurve,
          approximateDistance, true);
       MatOfPoint points = new MatOfPoint( approximateCurve.toArray() );
       Rect rectangle = Imgproc.boundingRect(points);

                if (rectangle.area() > max) {
                        Rect newRectangle = new Rect(rectangle.x,
                           rectectangle.y, rectangle.width, rectangle.
                           height);
                bitmap = Bitmap.createBitmap(rectangle.width, rectangle.
                   height, Bitmap.Config.ARGB_8888);
                copyMat = new Mat(mat, newRectangle); // mat - oryginalny
                   obraz w formacie Mat
                        max = rectangle.area();
        }
    }
Utils.matToBitmap(copy, bitmap);
image = bitmap;
}
```

Kod źródłowy 4.7: Kod programu, odpowiadający za wykrycie prostokątów przy użyciu biblioteki Open CV

## 4.4 Przetwarzanie wykrytego prostokąta

Jak wcześniej wspomniano wykryty prostokąt, z tekstowym znakiem drogowym, zostaje wycięty z oryginalnego zdjęcia, co znaczy, że jest to obraz, wciąż nie przetworzony. Należy go przetransformować do skali szarości i poddać progowaniu. Kolejne operacje są już ściśle związane ze specyfiką aplikacji. Z uwagi na konkretne cechy jakie musi mieć ob-

raz, który będzie obsługiwany przez bibliotekę Tess4j sprawdza się, czy posiada on więcej koloru czarnego niż białego, zakłada się, że aby tekst był koloru czarnego, a tło koloru białego, to właśnie białych pikseli musi być więcej. Do tego celu wykorzystano funkcję *isMoreBlack*, której implementację przedstawiono w Kodzie źródłowym 4.8. Wylicza ona liczbę pikseli czarnych i białych, a następnie sprawdza, których jest więcej.

```
int blackCounter = 0;
int whiteCounter = 0;

for (int i = 0; i < image.getHeight(); i++) {
     for (int j = 0; j < image.getWidth(); j++) {
          int value = image.getPixel(j, i);
          if (value == Color.BLACK) {
              blackCounter++;
          } else {
              whiteCounter++;
          }
    }
}
    if (blackCounter > whiteCounter)
        return true;
    else return false;
```

Kod źródłowy 4.8: Kod programu, odpowiadający za sprawdzenie czy jest więcej czarnych pikseli

Jeżeli powyższa funkcja zwróci wartość *true*, należy odwrócić kolory w obrazie, w programie zajmuje się tym funkcja *revertColors* zaprezentowana w Kodzie źródłowym 4.9.

```
Bitmap reverted = Bitmap.createBitmap(width, height, image.getConfig());

for (int i = 0; i < height; i++) {
     for (int j = 0; j < width; j++) {
          int value = image.getPixel(j, i);
          if (value == Color.BLACK) {
              reverted.setPixel(j, i, Color.WHITE);
          } else {
              reverted.setPixel(j, i, Color.BLACK);
          }
    }
}
image = reverted;
```

Kod źródłowy 4.9: Kod programu, odpowiadający za odwrócenie kolorów w obrazie

Część znaków tekstowych drogowych znajdujących się na ulicach posiada też obwódkę, która czasem powoduje, że biblioteka Tess4j nie jest w stanie wykryć tekstu znajdującego się na znaku. Dlatego napisano funkcję, która sprawdza czy owa obwódka istnieje. Jest to funkcja *detectBorded*, której kod źródłowy został przedstawiony pod numerem 4.10.

```
for (int i = 1; i < height*0.1; i++) {
    for (int j = 1; j < width; j++) {
         if (image.getPixel(j, i) != image.getPixel(j-1,i-1))
             return true;
    }
}
```

```
return false;
```

Kod źródłowy 4.10: Kod programu, odpowiadający za sprawdzenie czy wykryty prostokąt otoczony jest obwódką w innym kolorze

W przypadku gdy funkcja *detectBorder* zwróci wartość *true*, obraz zostaje wycięty w taki sposób, aby obwódki się pozbyć. Po analizie znaków drogowych tekstowych, określone przedziały procentowe znaku w jakich obwódka może występować i na podstawie tych badań napisano funkcję *cutBorder*, Kod źródłowy 4.11.

```
int borderWidth = (int) (0.05*width);
int borderHeight = (int) (0.05*height);
int newWidth = width - 2*borderWidth;
int newHeight = height - 2*borderHeight;
Bitmap imageWithoutBorders = Bitmap.createBitmap(newWidth, newHeight, image
    .getConfig());
int i_b = 0;
for (int i = borderHeight; i < newHeight+borderHeight; i++) {
     int j_b = 0;
     for (int j = borderWidth; j < newWidth+borderWidth; j++) {
          imageWithoutBorders.setPixel(j_b++, i_b, image.getPixel(j, i));
     }
     i_b++;
}
image = imageWithoutBorders;
```

Kod źródłowy 4.11: Kod programu, odpowiadający za obcięcie obrazu o konkretną procentową wartość

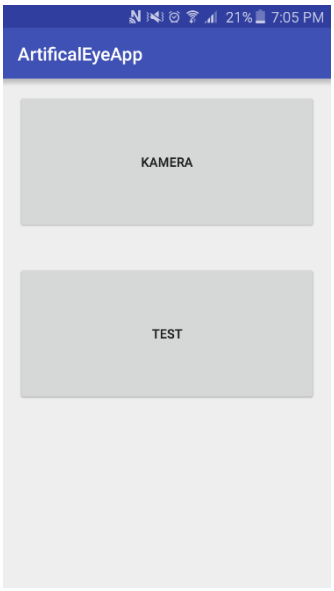
# Rozdział 5

# Działanie aplikacji i testy

## 5.1 Wymagania wstępne

Aplikacja działa na smartfonach z oprogramowaniem Android o wersji SDK nie starszej niż 17. Wersją najbardziej oczekiwaną jest wersja 23. Smartfon musi posiadać aparat oraz ponad 13Mb wolnej pamięci – zostanie na niej zapisany folder – tessdata, z plikiem zawierającym dane potrzebne bibliotece Tess4j do przetwarzania obrazu w języku polskim. Język telefonu powinien być ustawiony na ten w jakim oczekiwane jest czytanie komunikatów przez aplikację. Poniżej zawarto zrzuty ekranu robione urządzeniem mobilnym na którym działała aplikacja.

## 5.2 Widok główny




Przykład 5.1: Główny widok aplikacji
źródło: opracowanie własne

Jest to intuicyjny Widok z dwoma dużymi przyciskami. Przy uruchomieniu aplikacji, informuje ona użytkownika komunikatem głosowym o położeniu przycisków i ich przeznaczeniu: „Kliknij na górze ekranu aby przejść do kamery – Kliknij na dole ekranu aby

przejść do testów". Aplikacja, poza Widokiem głównym, posiada dwa Widoki dodatkowe: użytkownika oraz testowy. Można się do nich dostać po wciśnięciu w odpowiedni przycisk. Przycisk „Kamera" przekierowuje użytkownika do widoku z kamery smartfona, natomiast przycisk „Test" przekierowuje do Widoku testowego.

## 5.3 Widok docelowy

Widok docelowy przenosi użytkownika do kamery zainstalowanej na urządzeniu na którym działa aplikacja. W tym momencie użytkownik ma możliwość kliknięcia w ekran i zrobienia fotografii. W zależności od rodzaju telefonu, kamera może poprosić o potwierdzenie przesłania zdjęcia do dalszej przeróbki, o czym aplikacja poinformuje komunikatem głosowym, z dokładnym umiejscowieniem odpowiedniego przycisku na ekranie, zdjęcie też może zostać od razu przesłane.

## 5.4 Widok testowy


Przykład 5.2: Testowy widok aplikacji
źródło: opracowanie własne

Osoba testująca, klikając na wybrane zdjęcie sprawia, że zostaje ono poddane przetwarzaniu. W efekcie końcowym otrzymuje się rozpoznany znak, wycięty ze zdjęcia, jako obraz binarny – zaprezentowany przez dwa odcienie – czarny oraz biały, kolory odwrócone, odpowiednio tak aby tekst był czarny, a tło białe, czyli obraz jaki na wejście otrzymuje algorytm rozpoznający tekst. Pod obrazem zostaje wyświetlony tekst, którzy otrzymano w wyniku działania biblioteki Tess4j. Zostaje on przeczytany przez aplikację, użytkownik może odtwarzać go dowolną liczbę razy, klikając na przycisk „Powiedz". Poniżej, na zdjęciach numer 5.3, 5.4 oraz 5.5, zostały przedstawione wyniki działania aplikacji w trybie testowym.


(a) Obraz przetwarzany


(b) Obraz przetworzony

Przykład 5.3: Wynik działania aplikacji
źródło: opracowanie własne


(a) Obraz przetwarzany


(b) Obraz przetworzony

Przykład 5.4: Wynik działania aplikacji
źródło: opracowanie własne


(a) Obraz przetwarzany


(b) Obraz przetworzony

Przykład 5.5: Wynik działania aplikacji
źródło: opracowanie własne

## 5.5 Testy

Głównym problemem do rozwiązania było odpowiednie przetworzenie obrazu, aby biblioteka Tess4j sobie z nim poradziła. Testy zostały przeprowadzone w środwisku Intellij IDEA, a wyniki w postaci obrazów, zostały przedstawione poniżej.

Testowanie aplikacji zaczęto od najprostszych zdjęć znaków drogowych, gdzie wystarczyło je przekonwertować do skali szarości i poddać progowaniu, przykład przedstawiono na Przykładzie 5.6.


(a) Obraz przetwarzany (b) Obraz przetworzony

Przykład 5.6: Wynik działania algorytmu przetwarzania obrazu
źródło: opracowanie własne

Kolejną napotkaną trudnością był fakt, że na znakach tekst nie zawsze jest ciemniejszy od tła, więc należało przetestować algorytm i na takich obrazach, przykład na Przykładzie 5.7.


(a) Obraz przetwarzany (b) Obraz przetworzony

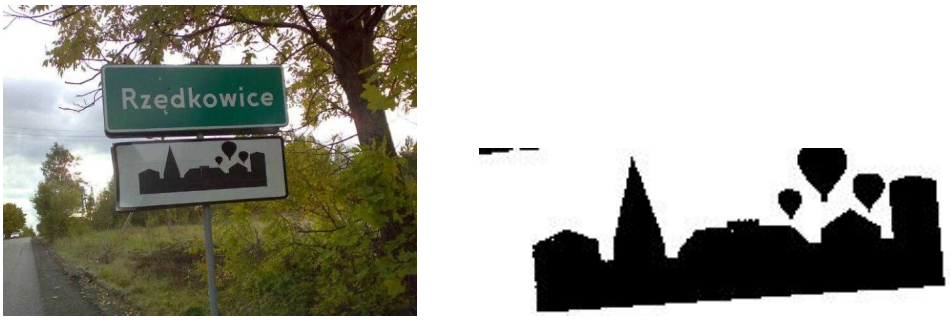
Przykład 5.7: Wynik działania algorytmu przetwarzania obrazu
źródło: opracowanie własne

Kolejno wypróbowano algorytm na znakach, które posiadały więcej kolorów lub były otoczone tłem jak na Przykładzie 5.8.


(a) Obraz przetwarzany (b) Obraz przetworzony

Przykład 5.8: Wynik działania algorytmu przetwarzania obrazu
źródło: opracowanie własne

Z uwagi na przyjęte założenia, dotyczące tego, że rozpoznawany znak jest zawsze największym prostokątem na zrobionej fotografii w przypadkach kiedy sąsiadują z nim inne znaki (Przykład 5.9), niekoniecznie tekstowe, algorytm zawodzi.




(a) Obraz przetwarzany (b) Obraz przetworzony

Przykład 5.9: Wynik działania algorytmu przetwarzania obrazu
źródło: opracowanie własne

# Rozdział 6

# Podsumowanie

Stworzona aplikacja i jej zaimplementowane funkcjonalności pokryły się z zamierzonym celem niniejszej pracy inżynierskiej. Środowisko Android Studio, w którym została napisana umożliwiło spełnienie wszystkich założeń projektowych, a także przeprowadzanie testów nie tylko na maszynie wirtualnej, ale również na urządzeniu mobilnym, czyli docelowym urządzeniu, na którym aplikacja ma pracować. Powstała prosta w obsłudze aplikacja, z interfejsem adekwatnym dla obranego użytkownika, jakim jest osoba niewidoma lub niedowidząca. Aplikacja ta korzysta z popularnych, prostych, ale skutecznych algorytmów przetwarzania obrazów, które zostały w powyższej pracy skrupulatnie opisane i przedstawione na tle innych możliwych rozwiązań, poza własną implementacją algorytmów, skorzystano także z dostępnych w sieci otwartych bibliotek - OpenCV oraz Tess4j. One również zostały omówione, przedstawiono ich zastosowania oraz przykładowe użycie.

Aplikacja jest napisana obiektowo, z możliwością rozszerzenia o kolejne funkcjonalności. Po prostu dodając kolejne Aktywności i Widoki. Napisano ją w języku Java, korzystając również z bibliotek natywnie napisanych w języku C++. Przy minimalnych zmianach, z uwagi na to, że biblioteki używane w Androidzie a te typowo związane z platformą Java, przy przetwarzaniu obrazów nieznacznie się różnią, aplikacja ta może pełnić funkcję aplikacji desktopowe do przetwarzania wczytanych obrazów na informację tekstową. Migracja programu na inne platformy, takie jak WindowsPhone, nie powinna być problematyczna ze względu na niebotyczne podobieństwo języków C# oraz Java.

# Bibliografia


[1] HowStuffWorks: *How BrailleTouch works* (dostęp: 3.12.2016)

URL: http://electronics.howstuffworks.com/gadgets/other-gadgets/braille-touch.htm

[2] R. Tadeusiewicz, P. Korohoda: *Komputerowa analiza i przetwarzanie obrazów*, Kraków: Wydawnictwo Fundacji Postępu Telekomunikacji, [1997] ('3)

[3] R. Tadeusiewicz, P. Korohoda: *Komputerowa analiza i przetwarzanie obrazów*, Kraków: Wydawnictwo Fundacji Postępu Telekomunikacji, [1997] ('3)

[4] jw. ('5)

[5] J.D. Cook: *Converting color to grayscale* (dostęp: 3.12.2016)

URL: http://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale

[6] Rafajłowicz E., Rafajłowicz W.: *Wstęp do przetwarzania obrazów przemysłowych*, Wrocław : Oficyna Wydawnicza Politechniki Wrocławskiej, [2010]. ('82)

[7] Wikipedia: *Histogram* (dostęp: 3.12.2016)

URL: https://pl.wikipedia.org/wiki/Histogram

[8] jw. ('83-'84)

[9] jw. ('85)

[10] jw. ('85)

[11] *IEEE Transactions on systems, man, and cybernetics*, vol. smc-9, no. 1, [1979] (dostęp: 3.12.2016)

URL: http://web-ext.u-aizu.ac.jp/course/bmclass/documents/otsu1979.pdf

[12] R. Jain, R. Kasturi, B. G. Schunck, *Machine Vision* McGraw-Hill, Inc., edycja 1, [1995] ('140)

[13] jw. ('145-146)

[14] *Gaussian Smoothing* (dostęp: 3.12.2016)

URL: http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm

[15] Android Developers: *AppCompatActivity* (dostęp: 3.12.2016)

URL: https://developer.android.com/reference/android/support/v7/app/AppCompatActivity.htm

[16] *Porównanie mobilnych systemów Android, iOS i Windows Phone 8* (dostęp: 3.12.2016)

URL: http://www.benchmark.pl/testy_i_recenzje/android-ios-windows.html

[17] B. Eckelo: *Thinking in Java edycja polska*, Gliwice: Helion, wydanie 4, [2006] . ('63-'64)

[18] jw. ('62)

[19] Android Developers: *TextToSpeech* (dostęp: 3.12.2016)

URL: https://developer.android.com/reference/android/speech/tts/TextToSpeech.html

[20] Android Developers: *Activity* (dostęp: 3.12.2016)

URL: https://developer.android.com/reference/android/app/Activity.html

[21] Android Developers: *View* (dostęp: 3.12.2016)

URL: https://developer.android.com/reference/android/view/View.html

[22] Android Developers: *Intent* (dostęp: 3.12.2016)

URL: https://developer.android.com/reference/android/content/Intent.html

[23] android4devs: *Wstęp do Intencji (Intents)* (dostęp: 3.12.2016 )

URL: http://www.android4devs.pl/2011/07/wstep-do-intencji-intents/

[24] Damian Chodorek: *Kurs Android (5)* (dostęp: 3.12.2016)

URL: http://damianchodorek.com/2015/02/12/kurs-android-intent-intencje-komunikacja-aktywnosci-5

[25] *The role of OCR in invoice processing* (dostęp: 3.12.2016)

URL: http://smart-soft.net/support/articles/automated-invoice-processing/the-role-of-ocr-in-invoice-processing.htm

[26] Wikipedia: *Automatic number plate tecognition* (dostęp: 3.12.2016)

URL: http://en.wikipedia.org/wiki/Automatic_number_plate_recognition

[27] *Tess4J - JNA wrapper for Tesseract* (dostęp: 3.12.2016)

URL: http://tess4j.sourceforge.net/

[28] E. Rafajłowicz, W. Rafajłowicz, A. Rusiecki: *Algorytmy przetwarzania obrazów i wstęp do pracy z biblioteką OpenCV*, Wrocław: Oficyna Wydawnicza Politechniki Wrocławskiej, [2010], ('23-'27)