Politechnika Warszawska

Cyfrowe Przetwarzanie Obrazu 2

DOKUMENTACJA

Projekt 1

Pr1 – Globalny próg na podstawie bimodalnego histogramu

Jarosław Affek
gr. 49

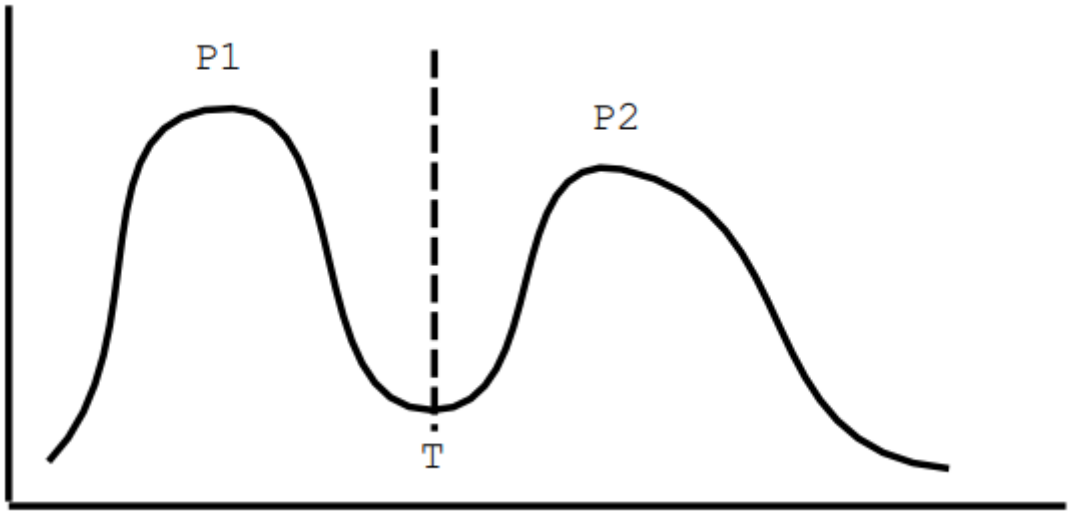
Warszawa 2018

1. Tabela z deklaracją zrealizowanych zadań

	Zadanie	Wariant	Punkty
Algorytm	Implementacja podstawowej wersji algorytmu	Bezbłędna	3,75
		Drobne uchybienia	3
		Błędy	2,25
		Poważne błędy	0,75
		Niepoprawna	0
	Parametryzacja rozwiązania	Pełna	2,25
		Częściowa	0,75
		Brak	0
	Implementacja innych realizacji, postaci		1
SUMA			7~7,5/7,5
Implementacja	Podział na metody – zasada pojedynczej odpowiedzialności		1,5
	Informatywne nazwy zmiennych		0,6
	Spójne nazewnictwo, język, styl		0,45
	Komentarze i przypisy		0,45
	Podział na interfejs i warstwę przetwarzania		0,75
	Kod bez powtórzeń		0,75
SUMA			4,5/4,5
Opis teoretyczny	Opis teoretyczny algorytmu		1,5
	Opis implementacji		0,75
	Opis wpływu parametrów na efekt działania		0,75
SUMA			3/3
SUMA CAŁKOWITA			14,5~15/15

2. Wstęp

Celem projektu było znalezienie progu binaryzacji dla obrazu, którego histogram ma charakter bimodalny. Przykład takiego histogramu przedstawiono na rys. 1.



Rysunek 1 [1]

Obrazy o takiej charakterystyce przedstawiają najczęściej obiekt, wyraźnie odróżniający się intensywnością od tła. W tego typu przypadku znalezienie progu, który oddzieliłby w procesie binaryzacji obszary o różnej jasności nie jest dla człowieka problemem. Znajduje się on w najniższym punkcie pomiędzy dwoma wysokimi pikami histogramu (na rys. 1 próg wyznacza pionowa linia T). Kolejnym krokiem jest zamiana wszystkich pikseli obrazu, których intensywność jest większa niż progowa na jedną wartość, natomiast pozostałe na drugą. Otrzymuje się w ten sposób obraz, w którym wartość każdego piksela można określić za pomocą jednego bitu. Analiza takiego histogramu przez program również nie jest skomplikowana i znalezienie progu jest trywialne, jednak ze względu na to, że w praktyce obrazy nigdy nie mają histogramu o tak ciągłej charakterystyce, należy zastosować specjalne algorytmy, które pomimo występowania szumów oraz lokalnych spadków wartości histogramu są w stanie poprawnie odszukać globalny próg histogramu. Algrytm ten ma zastosowanie do automatycznej analizy wizyjnej w różnego rodzaju procesach produkcyjnych, w których ważne jest np. określenie kształtu, wymiaru, powierzchni itp., a żeby to zrobić należy najpierw określić co na obrazie jest obiektem a co tłem.

3. Algorytm

- **Implementacja podstawowej wersji algorytmu**

Zadanie: Należy obliczyć histogram dla obrazu oraz zaimplementować algorytm binaryzacji z progiem obliczonym na podstawie bimodalnego histogramu. Operacja przeprowadzana będzie na obrazie w skali szarości. Należy przedyskutować skuteczność algorytmu (wskazać klasę obrazów dla których algorytm działa najlepiej).

Realizacja:

- Wczytany obraz jest konwertowany na skalę szarości (jeśli zachodzi taka potrzeba, czyli wczytany obraz jest kolorowy, a ponieważ w OpenCV polecenie imread w przypadku braku flagi CV_LOAD_IMAGE_GRAYSCALE każdy obraz zamienia na 3 kanałowy to operacja konwersji wykonywana jest za każdym razem) poprzez sumowanie i podział sumy składowych przez ich liczbę i wpisanie do jednokanałowego obrazu wyjściowego.

```
// conversion rgb to grayscale image
void convert_to_grayscale(const Mat & source_img, Mat & out_img)
{
	out_img = Mat(source_img.rows, source_img.cols, CV_8UC1);
	for (int i = 0; i < source_img.cols; i++)
		for (int j = 0; j < source_img.rows; j++)
			out_img.at<uchar>(j, i) = (source_img.at<Vec3b>(j, i)[0] +
source_img.at<Vec3b>(j, i)[1] + source_img.at<Vec3b>(j, i)[2]) / 3;
}
```

- Obliczany jest histogram obrazu w skali szarości. Odbywa się to poprzez sumowanie pikseli o poszczególnych intensywnościach i wpisanie ich do wektora. Numer elementu wektora odpowiada jasności (długość wektora dla typowego obrazu to 256 elementów (0 - 255)), natomiast wartości wpisane do poszczególnych elementów wektora to liczba pikseli o danej intensywności w obrazie.

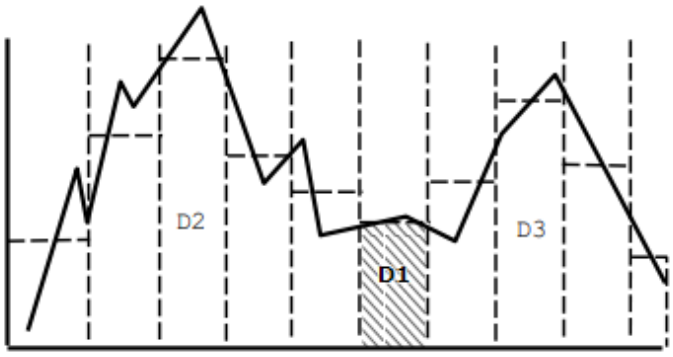
```
// calculate histogram of the image: intensity - the number of the vector element,
number of pixels as a value in each element
void calculate_histogram(vector<int> & hist, const Mat & source_img)
{
	for (int i = 0; i < source_img.cols; i++)
		for (int j = 0; j < source_img.rows; j++)
			hist[source_img.at<uchar>(j, i)] += 1;
}
```

Dzięki parametryzacji w kolejnym kroku programu możliwy jest podział histogramu na większe bloki (w celu np. uśrednienia lub zniwelowania drobnych skoków wartości histogramu). Parametryzacja omówiona będzie w kolejnym punkcie. Po opcjonalnym podziale histogramu uruchamiana jest funkcja wyliczająca próg binaryzacji. Ze względu na skomplikowanie algorytmu proces ten jest rozłożony na kilka powiązanych ze sobą funkcji. Do wyznaczenia progu zastosowano algorytm opisany w materiałach Uniwersytetu w Utah [1], który umożliwia poprawne dobranie progu nawet przy nieidealnie bimodalnym histogramie, który posiada szumy lub więcej niż jedno minimum lokalne.

Algorytm składa się z następujących kroków:

- wybranie dzielnika histogramu - podział histogramu na kilka tzw. podregionów – ich wysokość jest wyliczona jako średnia wysokości słupków wejściowego histogramu (rys. 2)
- kolejne działanie wykonywane są w pętli aż do momentu spełnienia określonego warunku, który oznacza, że znaleziony próg jest tym odpowiednim (warunkiem tym może być dojście dzielnika do liczby części, z których składał się pierwotny histogram)
    - zwiększenie dzielnika histogramu
    - podział histogramu przez dzielnik na mniejsze podregiony
    - wyszukanie wszystkich podregionów, które są lokalnymi minimami (otoczone przez podregiony o większej liczbie pikseli)
    - wybranie najbardziej znaczącego lokalnego minimum zgodnie z zasadą, że najbardziej znaczącym podregionem jest ten, dla którego wynik następującego działania jest **największy**:
    |najwyższy podregion po lewej – lokalne minimum|+|najwyższy podregion po prawej – lokalne minimum | → wg rys. 2: |D2-D1+|D3-D1|
    - znalezienie progu binaryzacji jako numeru najbardziej znaczącego lokalnego minimum, przeskalowanego do wartości intensywności (0 - 255)

Rysunek 2 i 3 przedstawiają algorytm w postaci ilustracji oraz pseudokodu.



Rysunek 2 [1]

```
Scan the image to get its histogram;
setup the initial number to divide the histogram;
do {
increase number to divide the histogram;
divide the histogram into subregion;
find the local minimum subregions;
find the most significant local minimum subregion;
find the threshold;
} while ( error(threshold, last_threshold)>ERROR);
```

Rysunek 3 [1]

Algorytm ten sprawia, że każde przejście pętli powoduje wyznaczenie nowego progu z coraz większą dokładnością, ponieważ histogram jest dzielony na coraz więcej „słupków”, porównywanie progu wyznaczonego z poprzedniego przejścia pętli do obecnego pozwala określić czy różnica między nimi jest na tyle mała, że można zakończyć wywoływanie pętli. Aby mieć pewność, że próg będzie wyznaczony z możliwie największą dokładnością można określić, że pętla będzie się wykonywać do momentu gdy histogram będzie podzielony na tyle części ile oryginalny (w przypadku standardowego histogramu – na 256). Poniżej przedstawiona jest implementacja algorytmu. Pierwsza funkcja (threshold_calculation) realizuje opisaną powyżej pętlę, jednak aby nie realizowała ona zbyt dużej liczby zadań wydzielone zostały kolejne dwie funkcje.

```
// loop that looks for the best match of the threshold to the histogram - implemented
algorithm
int threshold_calculation(vector<int> source_histogram)
{
	int hist_divisor = 2, threshold_previous = -1, threshold = -1, min_subregion = -
1;
	do
	{
		threshold_previous = threshold;
		hist_divisor *= 2;
		if (hist_divisor > source_histogram.size()) break;
		vector<int> div_histogram(hist_divisor);
		divide_histogram(source_histogram, div_histogram, hist_divisor);
		min_subregion = find_min_subregion(div_histogram, hist_divisor);
		threshold = 256 / (2 * hist_divisor) + min_subregion * (256 /
hist_divisor) - 1;
	} while (hist_divisor <= source_histogram.size());
	if (threshold < 0) threshold = 0;
	return threshold;
}
```

Funkcja (find_min_subregion) wyszukuje współrzędne (numer) podregionu określanego mianem najbardziej znaczącego minimum lokalnego zgodnie ze wzorem opisanym wyżej.

```
// algorithm: finding the index of the histogram subregion with the smallest average
number of pixels between 2 subregions with the biggest average number of pixels
int find_min_subregion(vector<int> hist, int divisor)
{
	int minimum_subregion = -1, max_left_subregion = -1, max_right_subregion = -1,
max_height_difference = -1;
	for (int i = 1; i < divisor - 1; i++)
		if (hist[i] < hist[i - 1] && hist[i] < hist[i + 1])
		{
			max_right_subregion = i + 1;
			max_left_subregion = i - 1;
			for (int j = i + 1; j < divisor - 1; j++)
				if (hist[j + 1] > hist[max_right_subregion])
					max_right_subregion = j + 1;
			for (int j = i - 1; j > 0; j--)
				if (hist[j - 1] > hist[max_left_subregion])
					max_left_subregion = j - 1;
			if ((abs(hist[max_left_subregion] - hist[i]) +
abs(hist[max_right_subregion] - hist[i])) > max_height_difference)
			{
				max_height_difference = abs(hist[max_left_subregion] -
hist[i]) + abs(hist[max_right_subregion] - hist[i]);
				minimum_subregion = i;
			}
```

```
			}
		return minimum_subregion;
}
```

Funkcja (divide_histogram) dzieli histogram na zadaną liczbę bloków, z których każdy ma wartość średniej z bloków składowych.

```
// dividing histogram values into blocks and saving as new simpler histogram
void divide_histogram(vector<int> source_histogram, vector<int> & out_histogram, int
divisor)
{
	int sum = 0;
	for (int i = 0; i < divisor; i++)
	{
		for (int j = i * source_histogram.size() / divisor; j < (i + 1) *
source_histogram.size() / divisor; j++)
			sum += source_histogram[j];
		out_histogram[i] = (int)(sum / (source_histogram.size() / divisor));
		sum = 0;
	}
}
```

Następną funkcją jest funkcja binaryzująca – zamienia ona piksele o wartości powyżej progu na białe a poniżej na czarne.

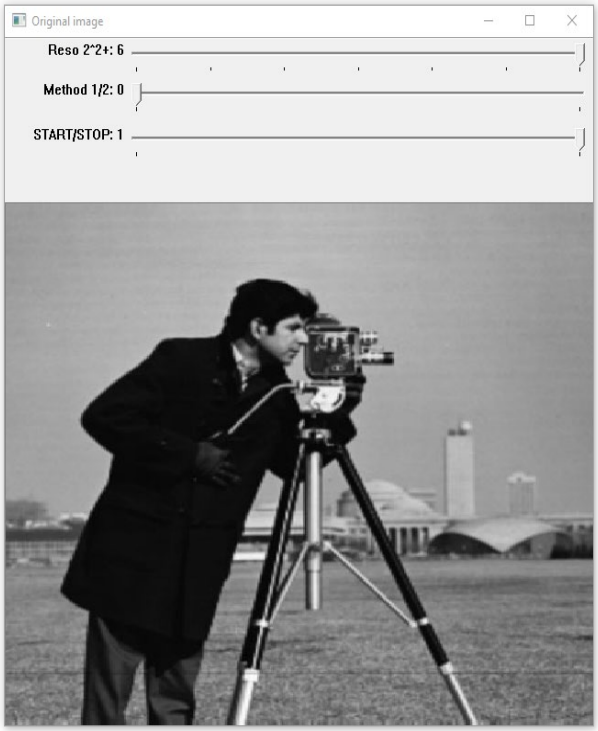
```
// creating a binarized image
void thresholding_image(int threshold, const Mat & source_img, Mat & out_img)
{
	out_img = Mat(source_img.rows, source_img.cols, CV_8UC1);
	for (int i = 0; i < source_img.cols; i++)
		for (int j = 0; j < source_img.rows; j++)
		{
			if (source_img.at<uchar>(j, i) > threshold)
				out_img.at<uchar>(j, i) = 255;
			else
				out_img.at<uchar>(j, i) = 0;
		}
}
```

Kolejnymi funkcjami wykonywanymi podczas cyklu programu są funkcję wyświetlające okna i obrazy oraz rysujące histogram, nie były one częścią polecenia jednakże dopiero narysowany histogram wraz z zaznaczonym oraz wypisanym progiem pozwala ocenić skuteczność algorytmu (funkcje te będą przedstawione w dalszej części dokumentacji). Wynik działania programu w postaci obrazów po zbinaryzowaniu oraz histogramu przedstawiony zostanie w punkcie „parametryzacja rozwiązania” gdyż będzie można przedstawić różnice w działaniu programu w zależności od różnych parametrów.

- **Parametryzacja rozwiązania**

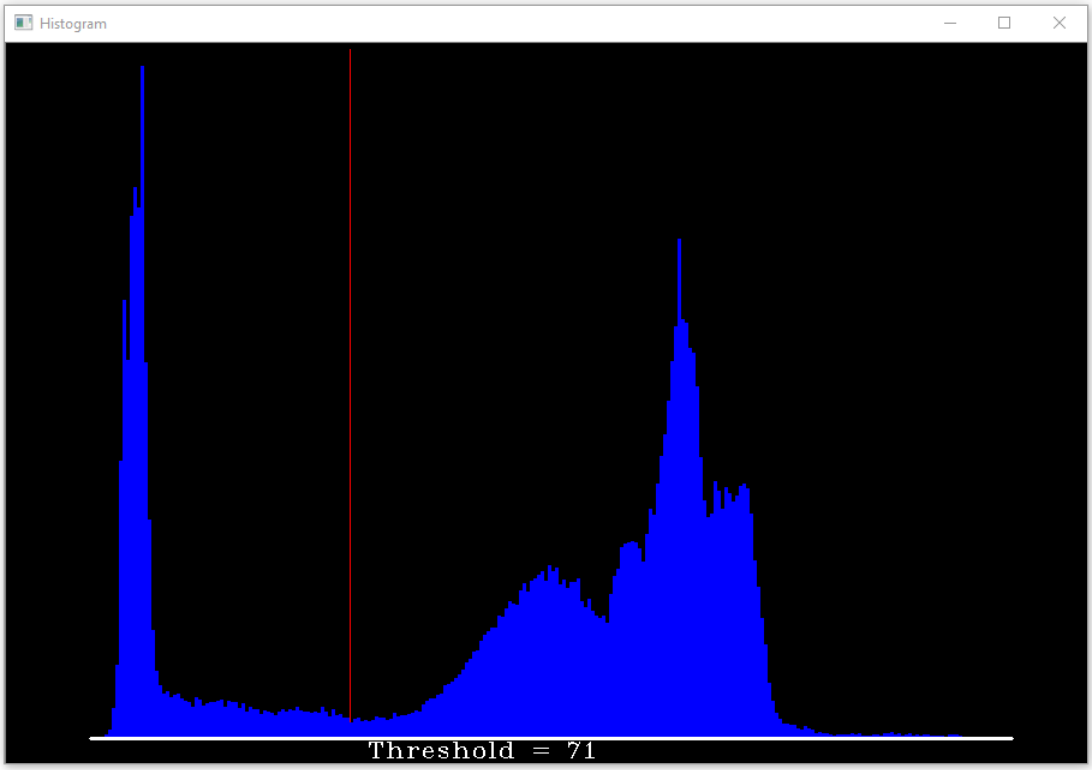
Jako, że algorytm wyznaczania progu nie posiada zmiennych i działa niezależnie od użytkownika to parametryzacja w tym projekcie została wykorzystana przy obliczaniu histogramu. Dzięki zastosowaniu GUI biblioteki OpenCV cała interakcja użytkownika z programem opiera się na 3 suwakach widocznych po otwarciu programu. Parametryzacja obliczania histogramu polega na możliwości doboru „rozdzielczości” histogramu obrazu – użytkownik może wybrać podział standardowy na 256 części (dla każdej intensywności jeden

pik histogramu) lub można histogram dzielić na mniej części, zmiana ta wpływa nie tylko na sam wygląd histogramu, ale również na wartość wyznaczonego progu, ponieważ poprzez podział histogramu na większe części możliwe jest np. usunięcie drobnych minimów lokalnych i ogólne uśrednienie zmian histogramu. W przypadku obrazów o wyraźnym histogramie bimodalnym prawdopodobnie spowoduje to mniejszą dokładność wyznaczenia progu, ale dla obrazów zaszumionych, z nieregularnym histogramem jest to być może jedyna szansa uzyskania mniej więcej prawidłowego progu. Drugim parametrem, który użytkownik może manipulować jest wybór algorytmu progowania. Do wyboru jest algorytm opisany powyżej lub drugi opisany w punkcie „implementacja innych realizacji". Trzeci suwak służy do uruchamiania programu, po ustawieniu 2 pierwszych suwaków w żądane pozycje po przełączeniu w pozycję START program wylicza i rysuje histogram wraz z wypisanym i zaznaczonym progiem oraz wyświetla obraz po zbinaryzowaniu.
Efekt pracy programu dla podstawowego algorytmu oraz standardowego 256 elementowego histogramu przedstawiają rys. 4, 5, 6:



Rysunek 4

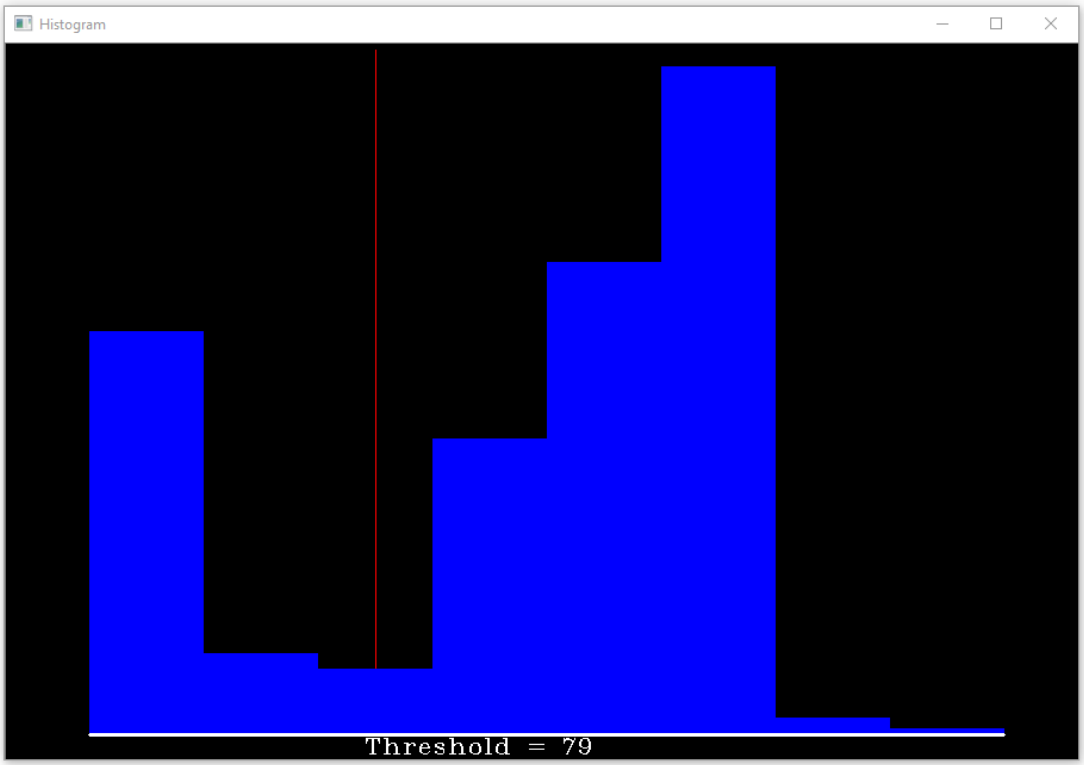
Rysunek 5



Rysunek 6

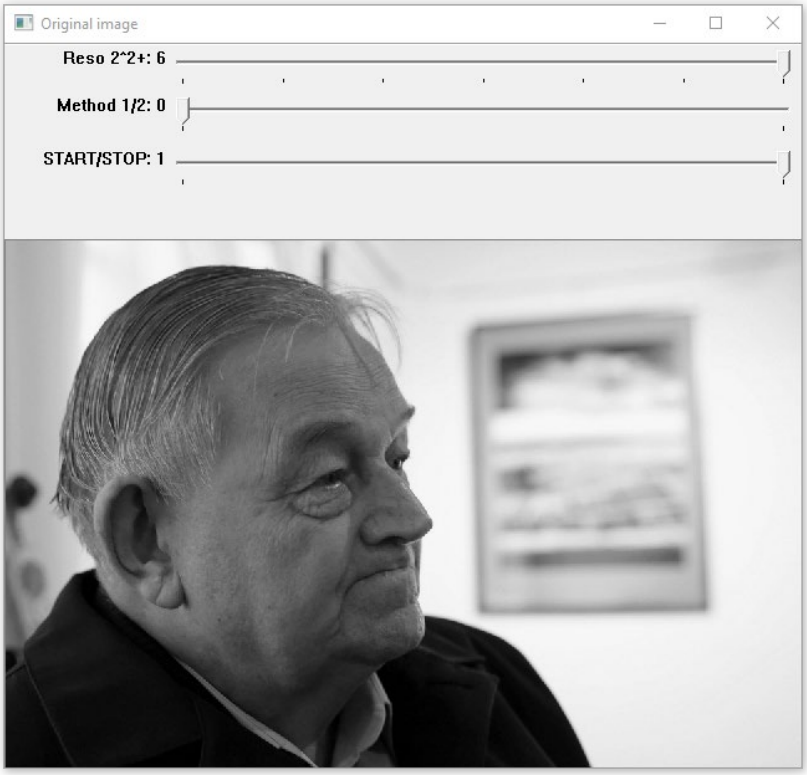
Ten sam obrazek i ten sam algorytm, ale przy podziale histogramu na 8 części przedstawiają rys. 7, 8.

Rysunek 7



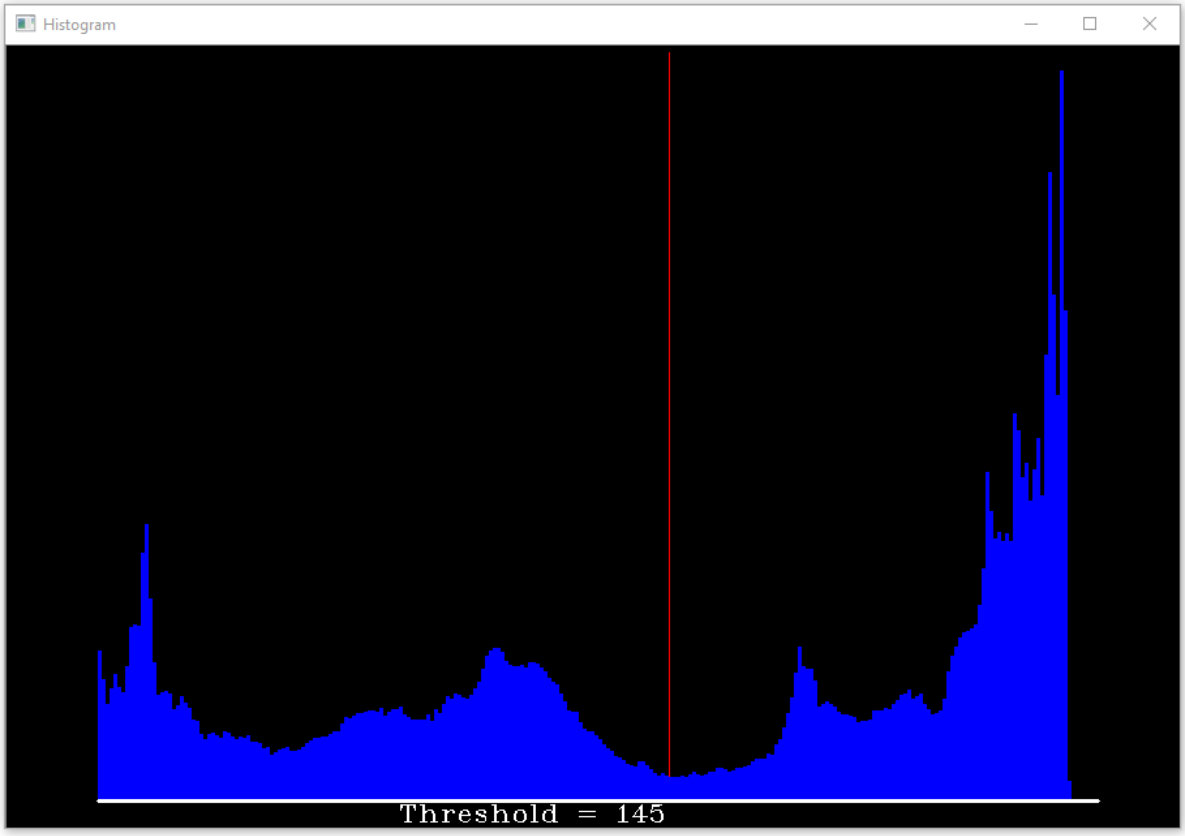
Rysunek 8

Jak widać zmiana liczby elementów histogramu ma wpływ na wyznaczony próg (71 i 79). Dla porównania rys. 9, 10, 11 przedstawiają obraz z histogramem bardziej odbiegającym od bimodalnego, jednak jak widać algorytm radzi sobie bardzo dobrze.



Rysunek 9

Rysunek 10



Rysunek 11

- **Implementacja innych realizacji, postaci**

Jako drugi algorytm zastosowany został tzw. „Balanced histogram thresholding” jest to znacznie prostszy algorytm polegający na „ważeniu” histogramu. Obierany jest środkowy punkt histogramu (dla histogramu 0 – 255 jest to punkt 127), następnie sumowane są wszystkie piksele po lewej stronie oraz oddzielnie – po prawej. Punkt środkowy jest punktem, do którego przyłożone są siły (siły są zależne od liczby pikseli po lewej i prawej stronie), jeżeli po którejś ze stron jest większa suma pikseli to punkt środkowy przesuwa się w przeciwną stronę o jeden słupek, a ostatni słupek z histogramu po „cięższej stronie” jest usuwany – zmienia się przez to rozkład pikseli po obu stronach od środkowego punktu. Operacją jest powtarzana do momentu gdy histogram składa się z jednego słupka – którego współrzędne określają próg. Pseudokod algorytmu przedstawiony jest na rys. 12 [2].

$$
\begin{aligned}
&I_m \leftarrow \tfrac{I_s+I_e}{2}\\
&W_l \leftarrow \textstyle\sum_{i=I_s}^{I_m} f_i\\
&W_r \leftarrow \textstyle\sum_{i=I_{m+1}}^{I_e} f_i\\
&\textbf{while } I_s \neq I_e\\
&\quad \textbf{do} \begin{cases}
\textbf{if } W_r > W_l\\
\quad \textbf{then} \begin{cases}
W_r \leftarrow W_r - f_{I_e}\\
I_e \leftarrow I_{e-1}\\
\textbf{if } \tfrac{I_s+I_e}{2} < I_m\\
\quad \textbf{then} \begin{cases}
W_l \leftarrow W_l - f_{I_m}\\
W_r \leftarrow W_r + f_{I_m}\\
I_m \leftarrow I_{m-1}
\end{cases}
\end{cases}\\
\textbf{else if } W_l \geq W_r\\
\quad \textbf{then} \begin{cases}
W_l \leftarrow W_l + f_{I_s}\\
I_s \leftarrow I_{s+1}\\
\textbf{if } \tfrac{I_s+I_e}{2} > I_m\\
\quad \textbf{then} \begin{cases}
W_l \leftarrow W_l + f_{I_m+1}\\
W_r \leftarrow W_r - f_{I_m+1}\\
I_m \leftarrow I_{m+1}
\end{cases}
\end{cases}
\end{cases}\\
&\textbf{return } (I_m)
\end{aligned}
$$

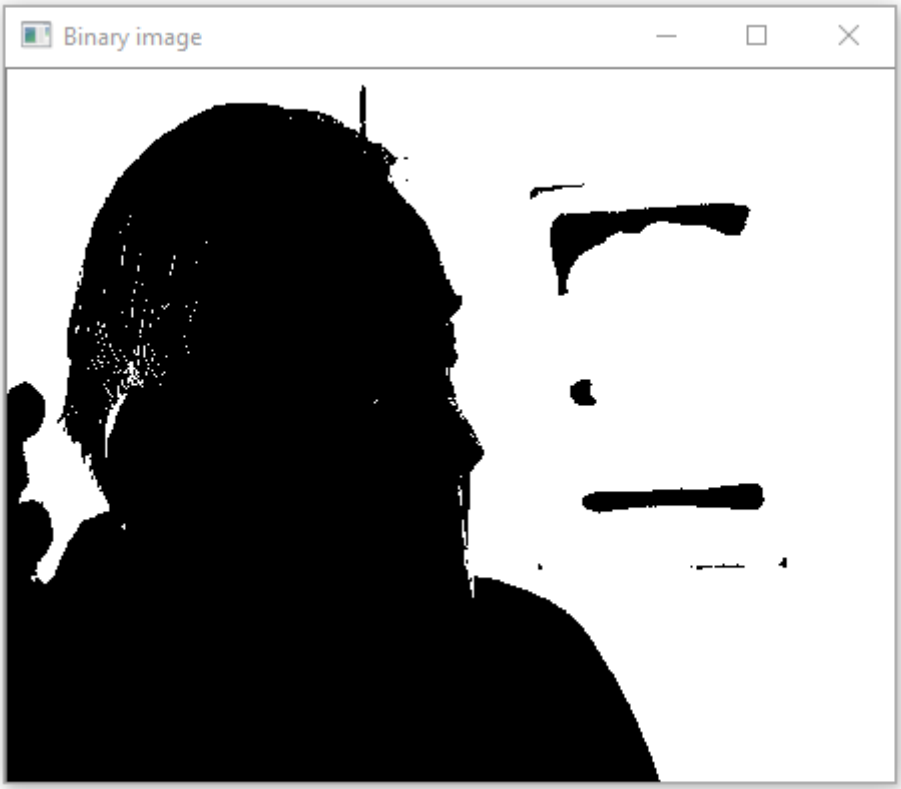
Rysunek 12 [2]

Implementacja algorytmu znajduje się poniżej:

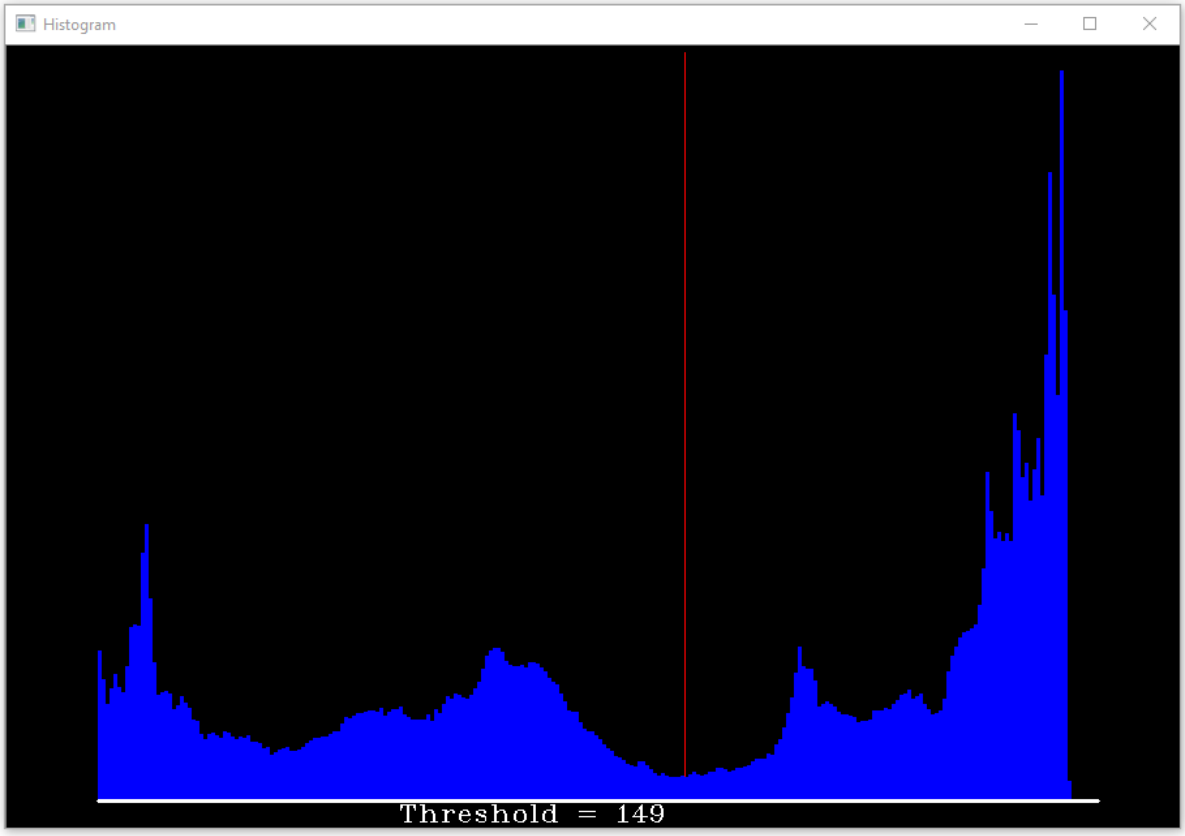
```
//an alternative algorithm for determining the threshold: Balanced histogram
thresholding
int balanced_thresholding(vector<int> source_histogram)
{
	int right_end = source_histogram.size() - 1, left_end = 0;
	int center_weight = (int)((right_end + left_end) / 2);
	int sum_left = 0, sum_right = 0;
	for (int i = 0; i < center_weight + 1; i++)
		sum_left += source_histogram[i];
	for (int i = center_weight + 1; i < right_end + 1; i++)
		sum_right += source_histogram[i];
	while (left_end <= right_end)
	{
		if (sum_right > sum_left)
		{
			sum_right -= source_histogram[right_end--];
			if (((right_end + left_end) / 2) < center_weight)
			{
				sum_right += source_histogram[center_weight];
				sum_left -= source_histogram[center_weight--];
			}
		}
		else
			if (sum_left >= sum_right)
			{
				sum_left -= source_histogram[left_end++];
				if (((left_end + right_end) / 2) > center_weight)
				{
					sum_left += source_histogram[center_weight + 1];
					sum_right -= source_histogram[center_weight + 1];
					center_weight++;
```

```
                        }
                    }
        }
        return (256 / (2 * source_histogram.size()) + center_weight * (256 /
source_histogram.size()) - 1);
}
```

Wzór przy zwracanej wielkości wynika z konieczności przeskalowania progu z numeru słupka do liczby odpowiadającej zakresowi 0- 255. Ten algorytm przy bimodalnych histogramach o w miarę równomiernym rozłożeniu pikseli daje poprawne wyniki, jednak przy obrazach, w których występują bardzo wysokie lub niskie i „wąskie” skoki wartości histogramu (minima i maksima) czasami daje błędne wyniki. Poniżej na rys. 13, 14 porównanie z poprzednim algorytmem.



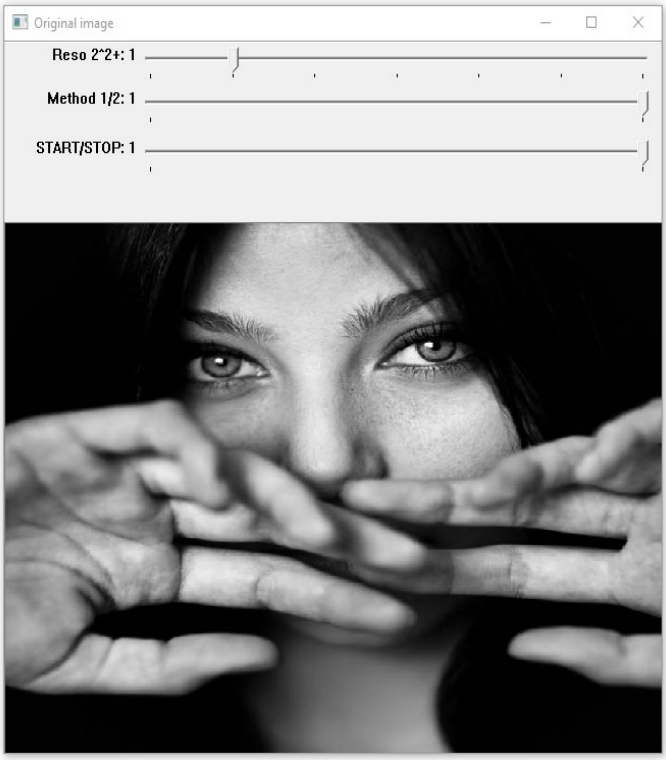
Rysunek 13



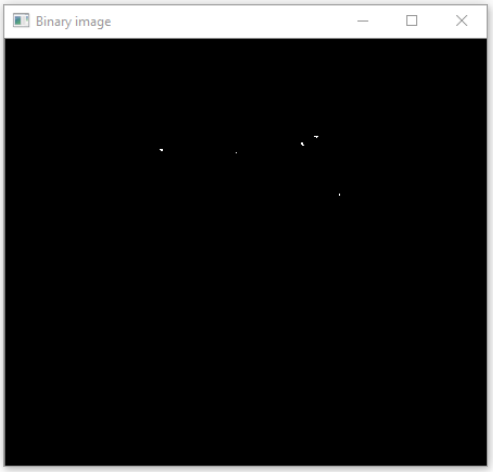
Rysunek 14

Jak widać nawet przy histogramie o dość skomplikowanym kształcie działanie algorytmu jest prawidłowe.

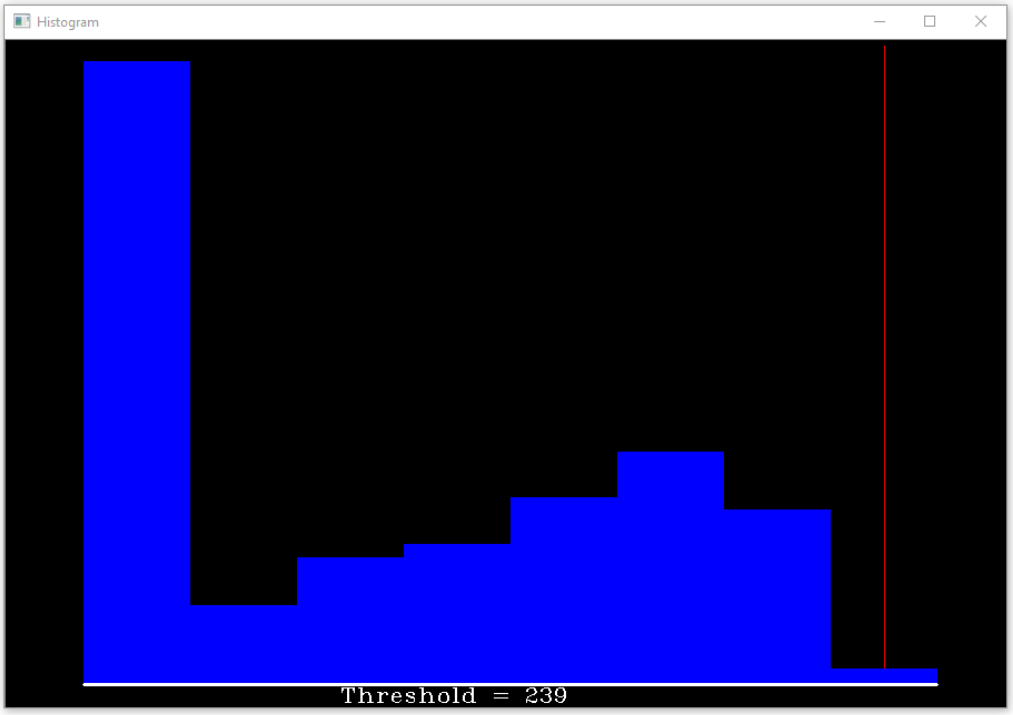
Poniżej przykład błędnego działania algorytmu „Balanced histogram thresholding” przy histogramie z dość dużymi różnicami w wartościach skrajnych słupków.



Rysunek 15

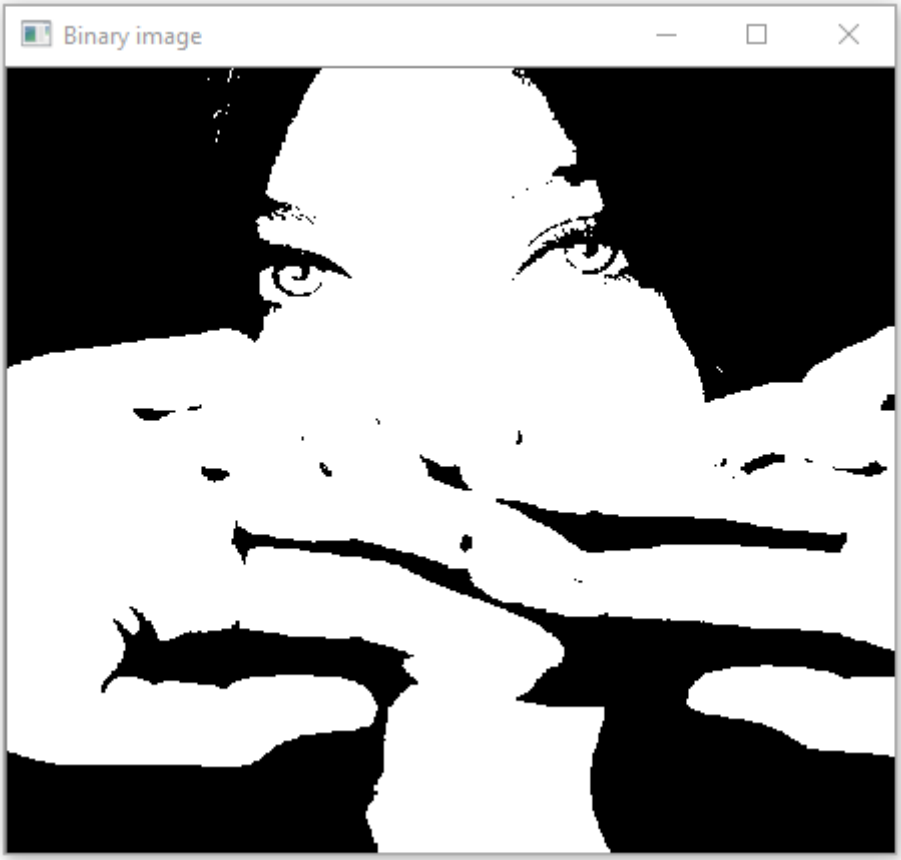


Rysunek 16

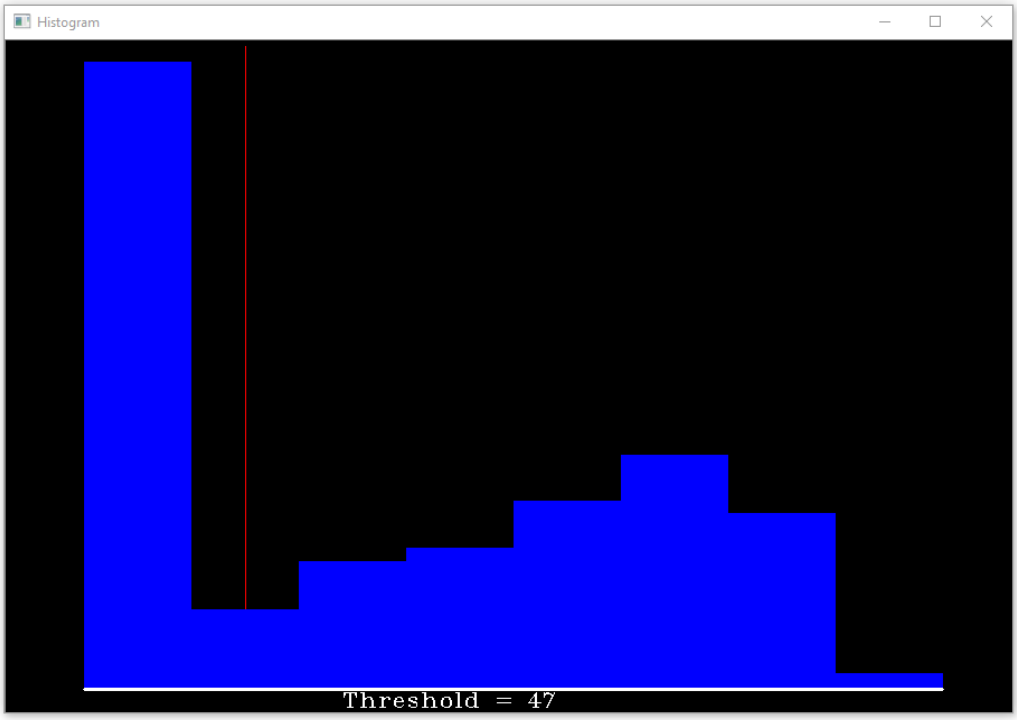


Rysunek 17

Dla porównania na rys. 18, 19 przedstawiono ten sam obraz oraz te same parametry histogramu oraz pierwszy z opisanych w tej dokumentacji algorytmów, który dał znacznie lepszy wynik.



Rysunek 18



Rysunek 19

4. Implementacja

- **Podział na metody – zasada pojedynczej odpowiedzialności**

Każda funkcja napisana w programie ma konkretne zadanie i wywołanie jej ma zawsze ma na celu wykonanie jednej i tej samej czynności. Są to np. funkcja konwertująca obraz na skalę szarości, funkcja wyświetlająca okna, funkcja rysująca histogram, są też funkcję obliczeniowe, takie jak dzielenie histogramu, obliczanie podregionu, jak również funkcje, które mają za zadanie jedynie wywołać w odpowiedniej kolejności inne funkcje (funkcje główne). Przykład – funkcja wyświetlająca okna:

```
// create and display image and histogram windows
void display_windows(const Mat & source_img, Mat & hist_bckg, Mat & binary_image)
{
		namedWindow("Histogram", WINDOW_AUTOSIZE);
		namedWindow("Binary image", WINDOW_NORMAL);
		imshow("Histogram", hist_bckg);
		imshow("Binary image", binary_image);
}
```

- **Informatywne nazwy zmiennych**

Każda zmienna ma jednoznacznie kojarzącą się nazwę i nazwa ta sugeruje przeznaczenie danej zmiennej. Również funkcje mają nazwy takie aby możliwie łatwo było się dowiedzieć co dana funkcja robi, nie analizując dogłębnie jej treści. Przykłady:

```
int hist_degree, start_calculation, algorythm_choice;
void calculate_histogram(vector<int> & hist, const Mat & source_img);
```

- **Spójne nazewnictwo, język, styl**

Język i styl są spójne, funkcje mają nazwy tworzone wg tych samych reguł (małe litery, słowa oddzielone podkreślnikami), cały kod wraz z komentarzami jest w języku angielskim, formatowanie pętli, funkcji, ustawienia klamer i wcięcia są takie same za każdym razem. Przykład:

```
// dividing histogram values into blocks and saving as new simpler histogram
void divide_histogram(vector<int> source_histogram, vector<int> & out_histogram, int
divisor)
{
	int sum = 0;
	for (int i = 0; i < divisor; i++)
	{
		for (int j = i * source_histogram.size() / divisor; j < (i + 1) *
source_histogram.size() / divisor; j++)
			sum += source_histogram[j];
		out_histogram[i] = (int)(sum / (source_histogram.size() / divisor));
		sum = 0;
	}
}
```

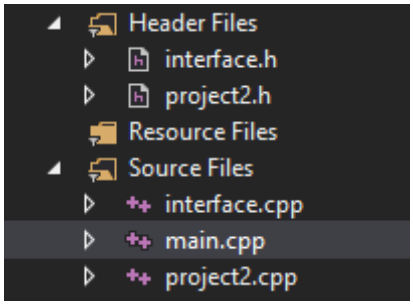
- **Komentarze i przypisy**

W kodzie oprócz nazwy funkcji informującej o jej działaniu znajdują się również komentarze, które jasno opisują każdą funkcję, trudniejsze zmienne oraz wszystkie wątpliwe momenty oraz zmienne globalne są wyjaśnione w komentarzach. Ponadto komentarze pełnią funkcję porządkowania kodu (oddzielają od siebie poszczególne bloki). Przykład:

```
extern int hist_degree, start_calculation, algorythm_choice;
extern bool gray_or_rgb;
// hist_degree                          define histogram resolution
// start_calculation                    starts and stops the program
// gray_or_rgb                          information about loaded image
// algorythm_choice                     define which algorythm is used

extern Mat loaded_img, gray_img;
//loaded_img                            the original photo loaded from the file
// gray_img                             original image converted to grayscale

//declared as global variables due to the gui environment requirements in opencv
(trackbars)

//*************************************************************************
// calculate histogram of the image: intensive - number of vector element, number of
pixels as a value in each element
void calculate_histogram(vector<int> & hist, const Mat & source_img);
//-------------------------------------------------------------------------
// dividing histogram values into blocks and saving as new simpler histogram
void divide_histogram(vector<int> source_histogram, vector<int> & out_histogram, int
divisor);
```

- **Podział na interfejs i warstwę przetwarzania**

Cały projekt podzielony jest na 5 plików: main.cpp, project2.cpp, project2.h, interface.cpp, interface.h. W pliku main jest jedynie wywołanie funkcji z pliku project2, natomiast wszystkie funkcje związane z wyświetlaniem i wczytywaniem danych znajdują się w osobnym pliku i są podzielone wg wykonywanych zadań. W plikach project2 znajduje się warstwa przetwarzania, a w interface – interfejs.



Rysunek 20

- **Kod bez powtórzeń**

Kod jest napisany w taki sposób aby nie było konieczne tworzenie dwa razy tej samej sekwencji wierszy kodu robiących to samo. Jeżeli jakiś element kodu jest wykonywany kilka razy (np. divide_histogram) to umieszczony został w oddzielnej funkcji i wywoływany jest w pętli.

5. Opis teoretyczny

- **Opis teoretyczny algorytmu**

Dokładny opis działania algorytmów zastosowanych w tym projekcie znajduje się w części „algorytm", wynika to z konieczności zaprezentowania teorii przed zacytowaniem kodu aby udowodnienie wykorzystania teorii w praktyce odbyło się dopiero po jej przedstawieniu. Z obserwacji działań obu algorytmów, dla różnych parametrów można zauważyć, że lepsze efekty otrzymano z pierwszego z nich, jednak jest on bardziej skomplikowany oraz trudniejszy do implementacji. Jednakże, jak widać na przedstawionych wyżej przykładowych obrazach obydwa algorytmy dają wyniki satysfakcjonujące dla histogramu w pełnej rozdzielczości, nawet wtedy gdy nie jest on idealnie bimodalny.

- **Opis implementacji**

Kod wraz z opisem implementacji znajduje się w części „algorytm - implementacja". W programie zastosowano dodatkowo funkcje rysujące, dzięki którym możliwa jest ocena prawidłowości działania algorytmów oraz programu.

```
// drawing the histogram on the black background
void draw_histogram(Mat & background, vector<int> px_values, int threshold)
{
	int max = 0;
	for (int i = 0; i < px_values.size(); i++)
		if (px_values[i] > max) max = px_values[i];
	float normalize = (float)(background.rows - 40) / max;
	line(background, Point(3*(threshold+1) + 70, background.rows - 21),
Point(3*(threshold+1) + 70, 5), Scalar(0, 0, 255), 1);
	for (int i = 0; i < px_values.size(); i++)
	{
		int height = px_values[i] * normalize;
		for(int j=0;j< 3*256/px_values.size();j++)
			line(background, Point(3*i*256/ px_values.size()+j + 70,
background.rows - 21), Point(3*i * 256 / px_values.size() + j + 70, background.rows -
21 - height), Scalar(255, 0, 0), 1, 8);
	}
	line(background, Point(70, background.rows - 21), Point(3 * (px_values.size() -
1) * 256 / px_values.size() + 3 * 256 / px_values.size() + 69, background.rows - 21),
Scalar(255, 255, 255), 2);
	char threshold_symbol[50];
	snprintf(threshold_symbol,100, "Threshold = %d", threshold);
	putText(background, threshold_symbol, Point(300, background.rows - 5),
FONT_HERSHEY_COMPLEX_SMALL, 1, Scalar(255, 255, 255), 1, 8);
}
```

Natomiast dzięki zastosowaniu suwaków, możliwe stało się wygodne i przede wszystkim szybkie zmienianie parametrów i podgląd efektów, bez konieczności wpisywania w konsolę tekstu i uruchamiania programu od nowa. Funkcja on_trackbar jest automatycznie wywoływana natychmiast po zmianie położenia suwaka START/STOP.

```
// function called when user move the START/STOP trackbar
void on_trackbar(int, void*)
{
	if (start_calculation == 1)
		if (gray_or_rgb == true)
			call(loaded_img);
		else
			call(gray_img);
	else
		destroy_windows();
}
```

Niestety GUI w bibliotece OpenCV jest bardzo ograniczone i nie występują tam przyciski. Również konieczność deklaracji globalnych zmiennych jest niepożądana, ale konieczna, ze względu na zasadę działania suwaków, które edytują bezpośrednio wartość zmiennej, która musi być znana w całej przestrzeni programu.

- **Opis wpływu parametrów na efekt działania**

Zmiana dzielnika histogramu powoduje zmianę „rozdzielczości" histogramu – im mniej bloków ma histogram tym bardziej jest on uśredniony i mniej widoczne są lokalne małe zmiany jego przebiegu. Dla pewnej grupy zdjęć z histogramem o dużej zmienności podział na większe bloki może być jedynym sposobem aby uzyskać mniej więcej prawidłowy próg binaryzacji z użyciem opisanych algorytmów. Więcej informacji o wpływie parametrów na algorytm znajduje się w części „algorytm". Obrazy przedstawiające efekty uzyskane dla różnych wartości parametrów umieszczane i opisywane były wcześniej, więc aby skrócić długość niniejszej instrukcji nie będą one tutaj powtarzane.

6. Bibliografia

[1] https://www.cs.utah.edu/docs/techreports/1994/pdf/UUCS-94-019.pdf

[2] https://www.researchgate.net/publication/221334567_Bi-Level_Image_Thresholding_-_A_Fast_Method