Politechnika Warszawska

Cyfrowe Przetwarzanie Obrazu 2

DOKUMENTACJA

Projekt 2

Detekcja i rozpoznawanie kart

Średnia harmoniczna

Jarosław Affek
gr. 49

Warszawa 2018

1. Tabela z deklaracją zrealizowanych zadań

	Zadanie	Punkty
Projekt rozwiązania	Przetwarzanie wstępne	1,5/1,5
	Uwydatnianie cech obiekt/tło	3,0/3,0
	Filtracja	1,5/1,5
	Segmentacja kart	6,0/6,0
	Przetwarzanie dodatkowe obszarów kart	1,5/1,5
	Segmentacja symboli	3,0/3,0
	Obliczenie cech	4,5/4,5
	Klasyfikacja obiektów	3,0/3,0
	Realizacja części a) polecenia	3,0/3,0
	Realizacja części b) polecenia	3,0/3,0
	suma:	**30,0/30,0**
Implementacja	Podział na metody – zasada pojedynczej odpowiedzialności	7,5/7,5
	Informatywne nazwy zmiennych	1,5/1,5
	Spójne nazewnictwo, język, styl	1,5/1,5
	Komentarze i przypisy	1,5/1,5
	Podział na interfejs i warstwę przetwarzania	3,0/3,0
	Kod bez powtórzeń	3,0/3,0
	suma:	**18,0/18,0**
Opis teoretyczny	Opis teoretyczny problemu	1,5/1,5
	Projekt ścieżki przetwarzania, schemat blokowy	3,0/3,0
	Opis problemów i ich rozwiązania	3,0/3,0
	Dobór algorytmów	3,0/3,0
	Ocena działania wybranych algorytmów	1,5/1,5
	suma:	**12,0/12,0**
	SUMA CAŁKOWITA:	**60,0/60,0**

Mnożnik punktów:

	MAX
Mnożnik	**1**
Przetwarzanie wstępne	1,5
Uwydatnianie cech obiekt/tło	3,0
Filtracja	1,5
Segmentacja kart	6,0
Przetwarzanie dodatkowe obszarów kart	1,5
Segmentacja symboli	3,0
Obliczenie cech	4,5
Klasyfikacja obiektów	3,0
Realizacja części a) polecenia	3,0
Realizacja części b) polecenia	3,0
SUMA	**30**

2. Wstęp

Celem projektu było stworzenie programu, którego zadaniem było rozpoznanie kart do gry (koloru i wartości – karty od 2 do 10). Zdjęć było 20 – 5 różnych scen (różnych zestawów kart), na każdej scenie 4 karty do gry przedstawione z pewnej perspektywy (obiektyw skierowany pod nieco innym kątem niż 90 w stosunku do płaszczyzny kart). Każda scena była przedstawiona na 4 zdjęciach:

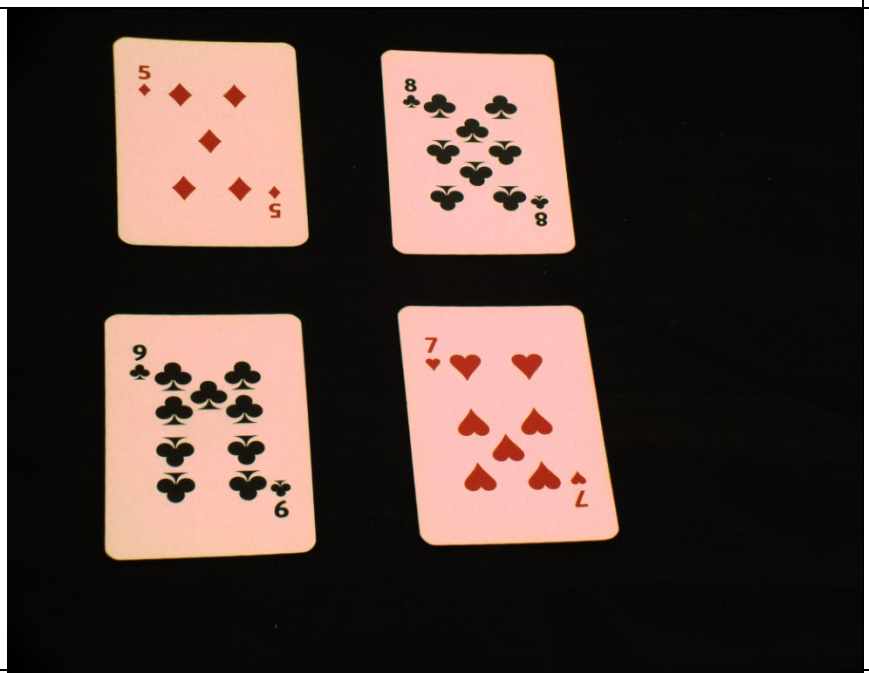
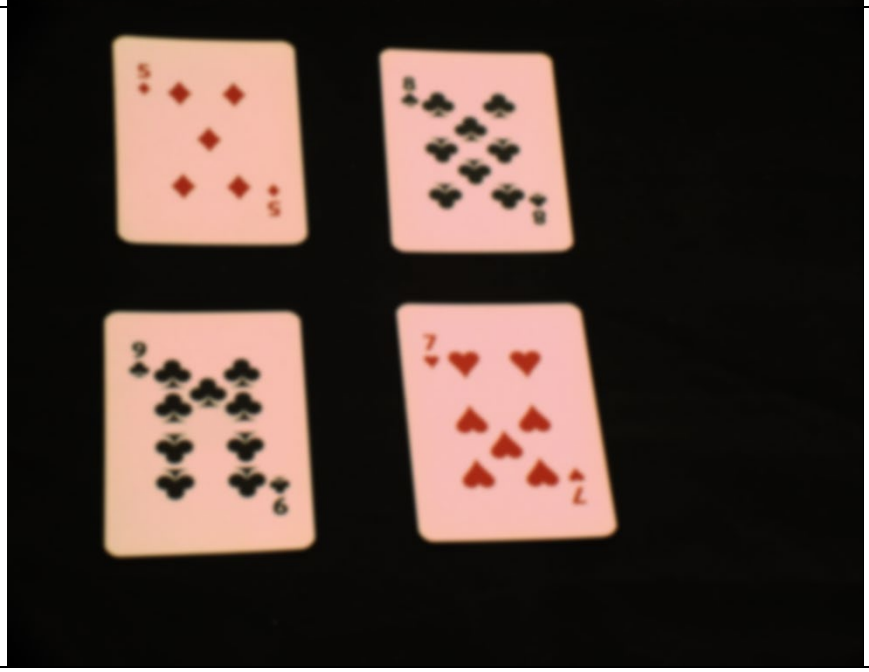
- ostrym bez filtrów
- rozmytym
- z nałożonym efektem gradientu
- z dodanym szumem

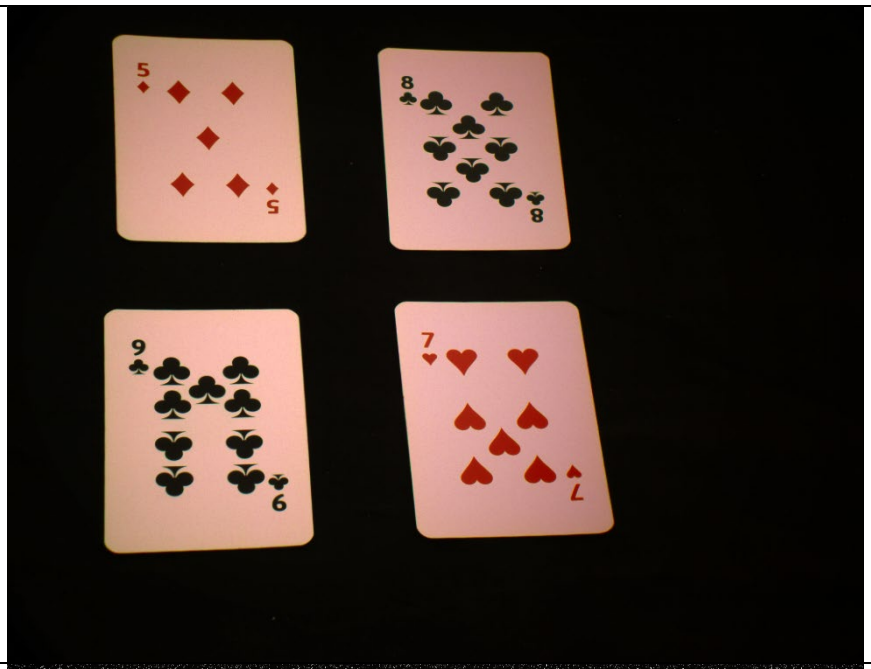
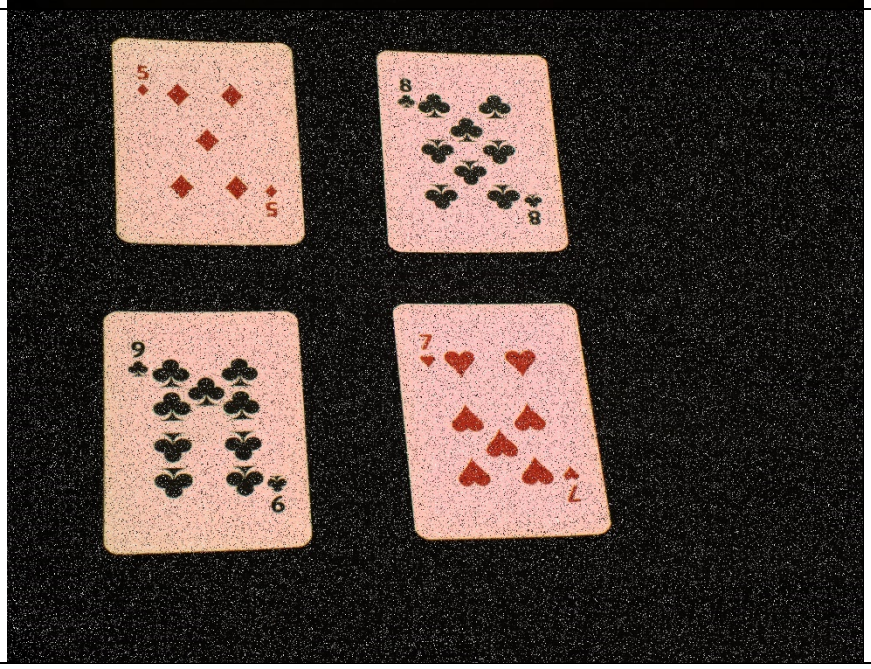
Zadaniem programu było rozpoznanie kart, najlepiej uniwersalnie, czyli dla każdej ze scen i jej wariantów ten sam algorytm.

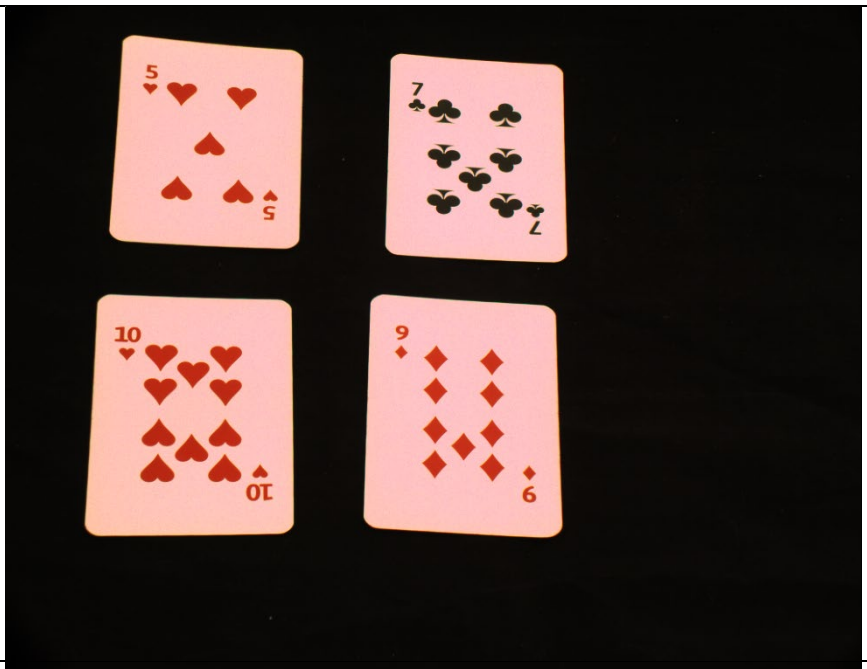
Dodatkowym zadaniem pobocznym było obliczenie średniej harmonicznej wszystkich znajdujących się w scenie symboli na kartach.

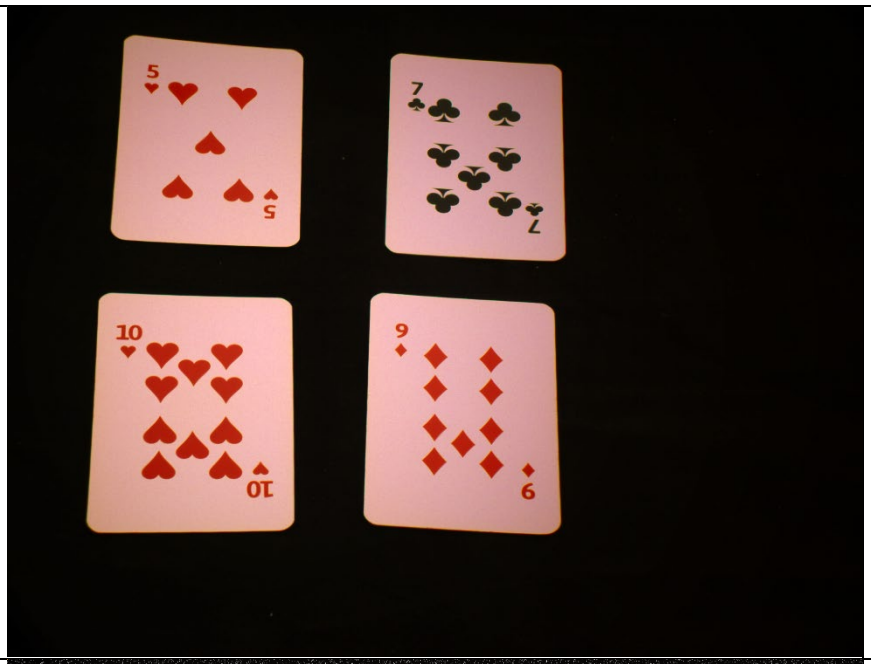
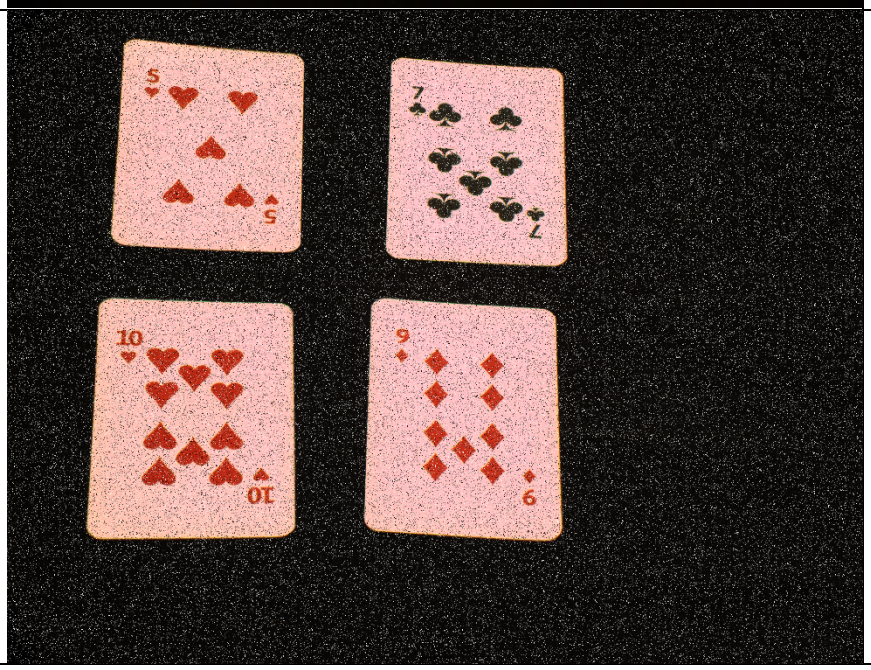
3. Efekt działania programu dla każdej ze scen

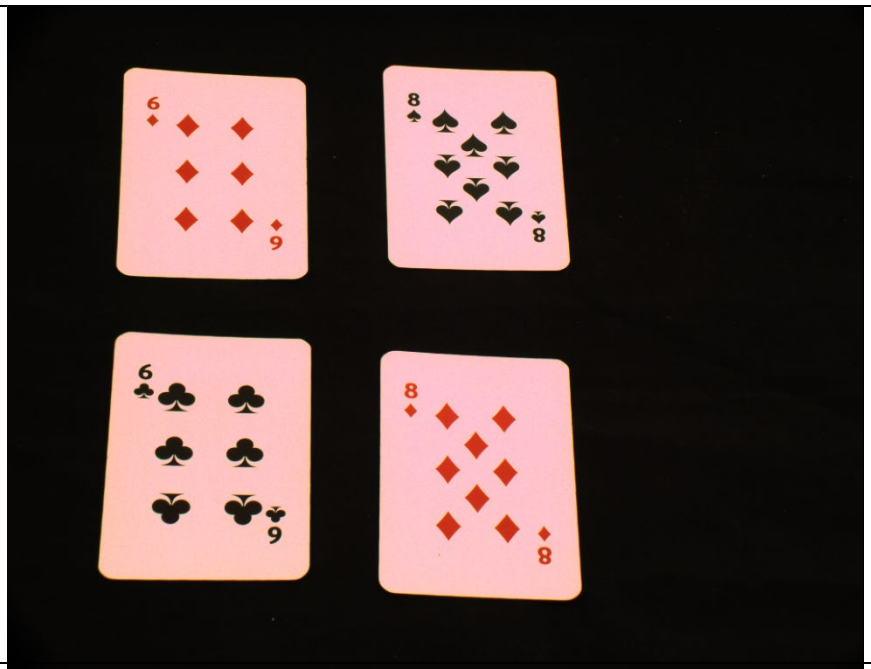
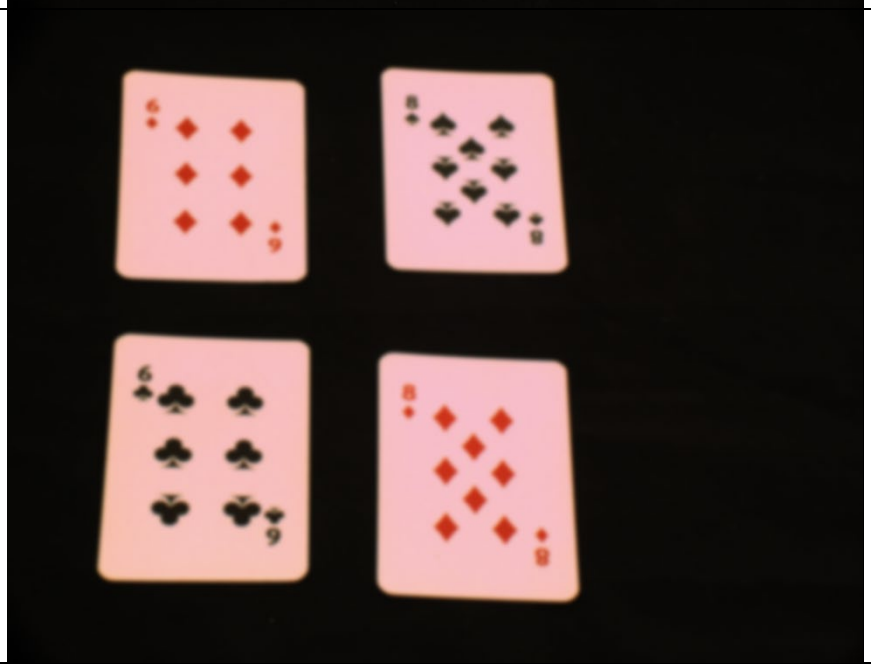
W celu szybkiego i sprawnego podglądu uzyskanych wyników program działa w ten sposób, że ładuje do pamięci operacyjnej wszystkie obrazy, przetwarza je a następnie wypisuje zbiorczy komunikat o wszystkich obrazach. Poniższa tabela przedstawia zdjęcie wejściowe i efekt działania programu (skopiowany tekst z konsoli). Pod tabelą umieszczony jest obraz rzeczywistego efektu generowanego przez program.

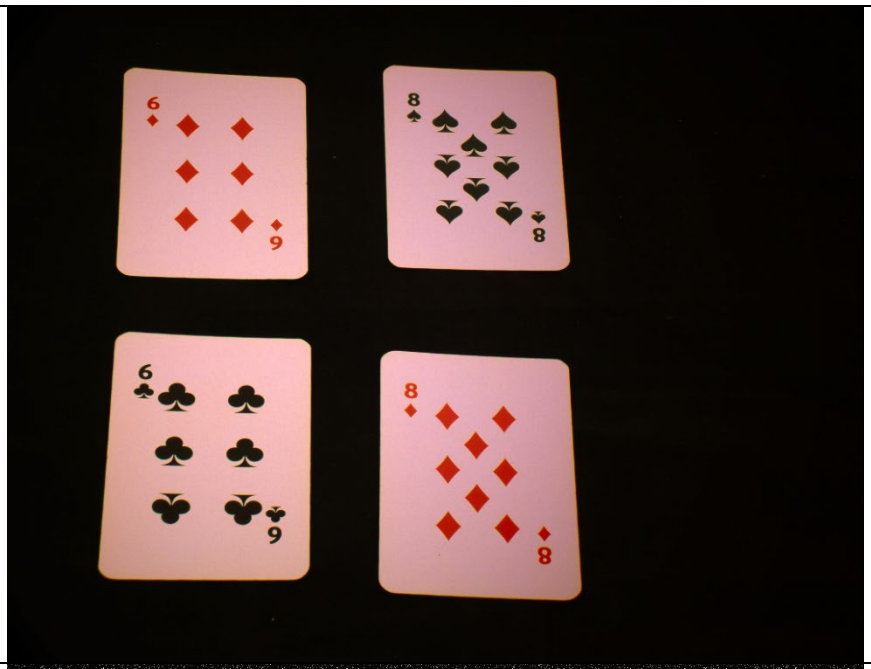
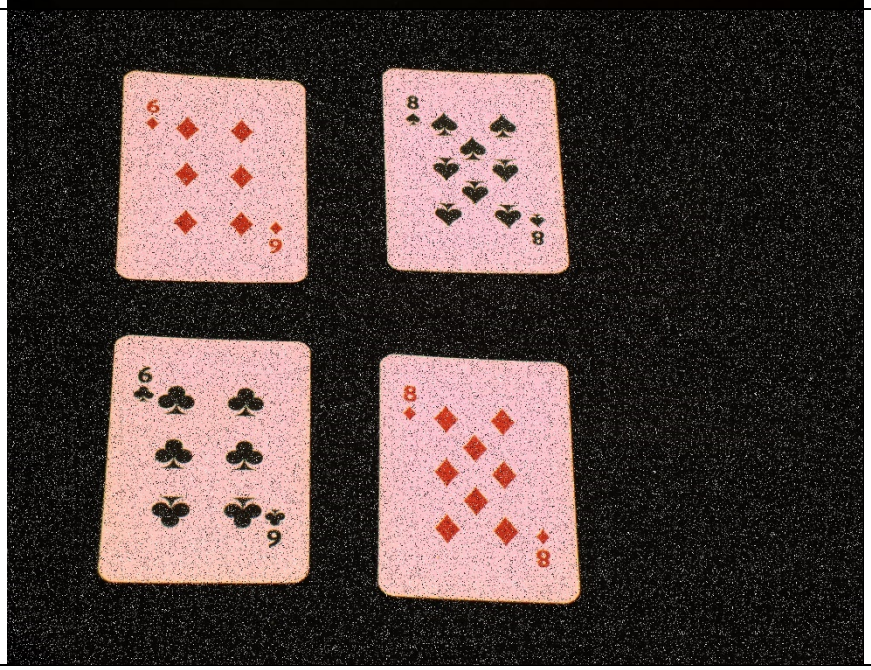
Nr	Dane wejściowe (obraz)	Dane wyjściowe (tekst)
1.		1.png 9 Clover 7 Heart 8 Clover 5 Tile Harmonic mean = 6.90884
2.		1_blurred.png 9 Clover 7 Heart 8 Clover 5 Tile Harmonic mean = 6.90884

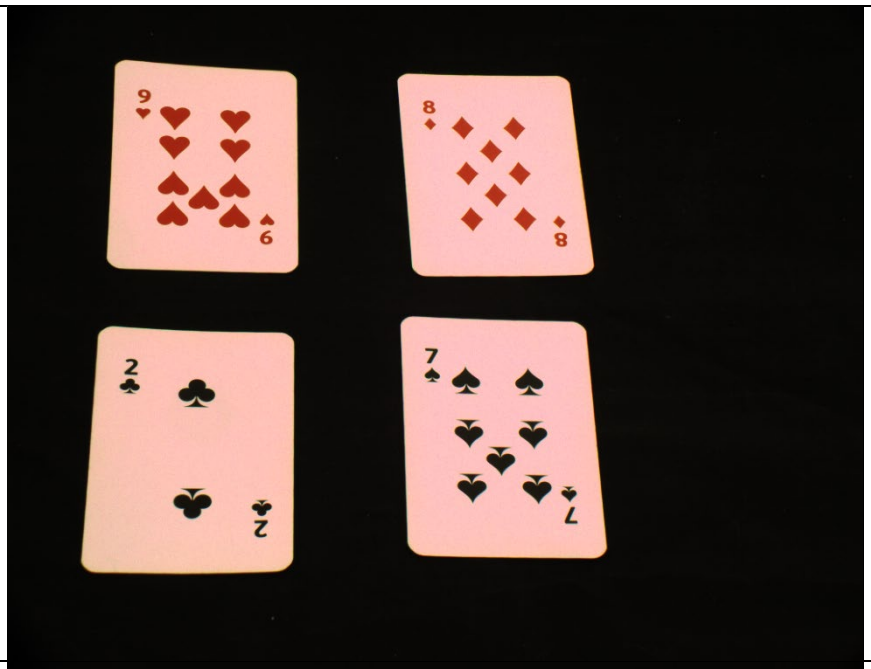
3.		1_gradient.png 9 Clover 7 Heart 8 Clover 5 Tile Harmonic mean = 6.90884
4.		1_noised.png 9 Clover 7 Heart 8 Clover 5 Tile Harmonic mean = 6.90884

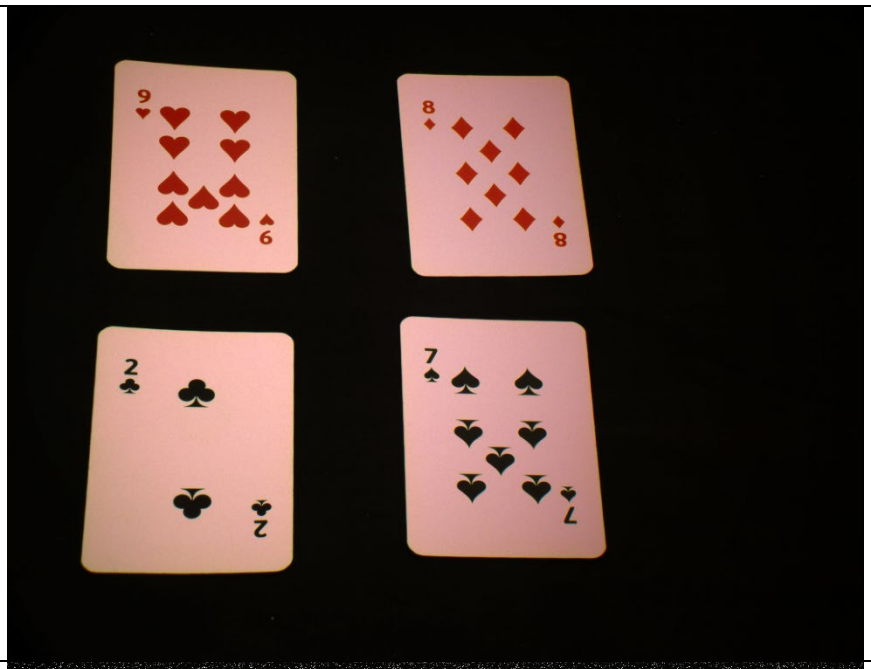
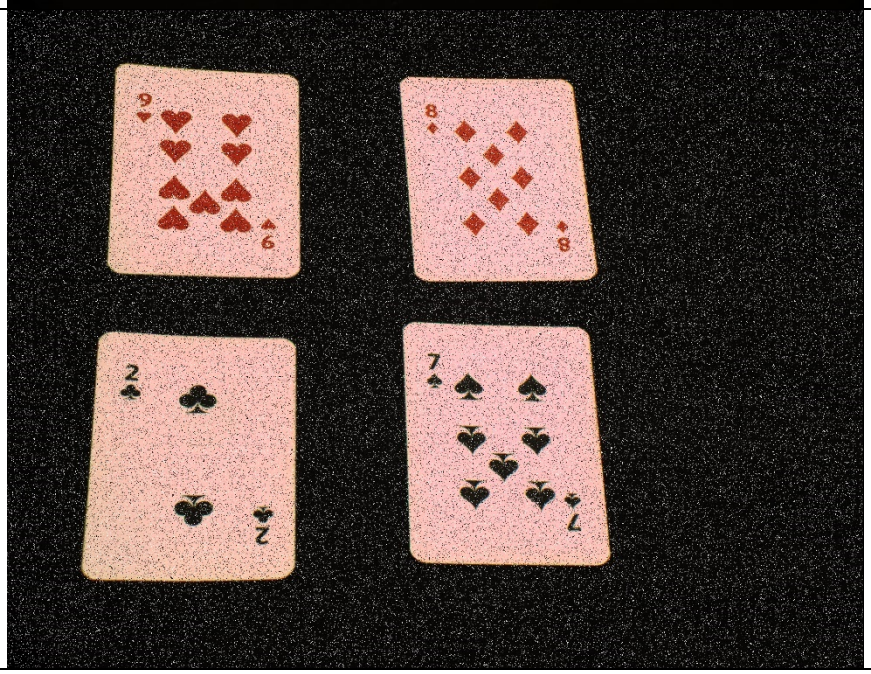
5.			2.png 9 Tile 10 Heart 7 Clover 5 Heart Harmonic mean = 7.22063
6.			2_blurred.png 10 Heart 9 Tile 7 Clover 5 Heart Harmonic mean = 7.22063

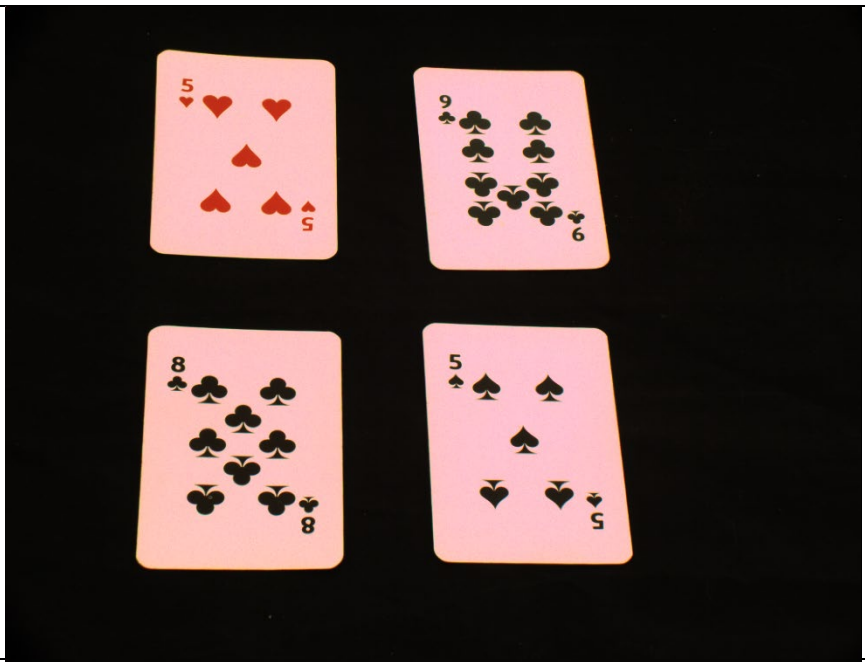
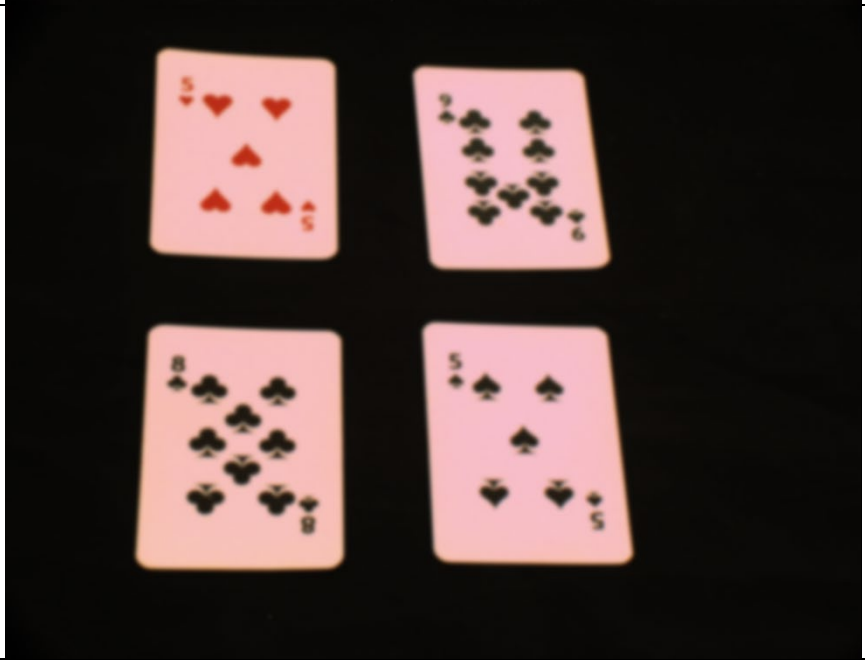
7.		2_gradient.png 9 Tile 10 Heart 7 Clover 5 Heart Harmonic mean = 7.22063
8.		2_noised.png 9 Tile 10 Heart 7 Clover 5 Heart Harmonic mean = 7.22063

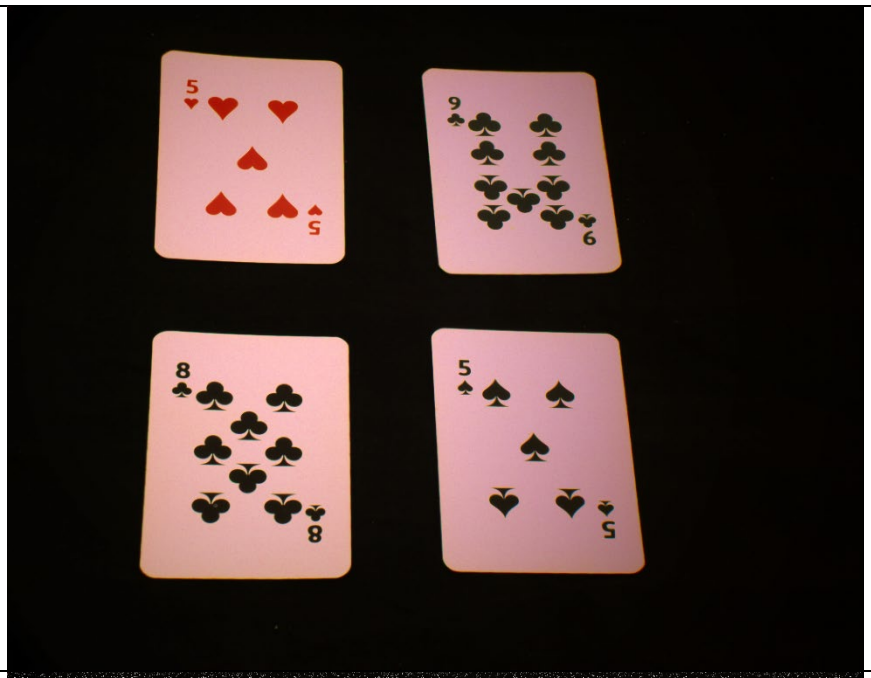
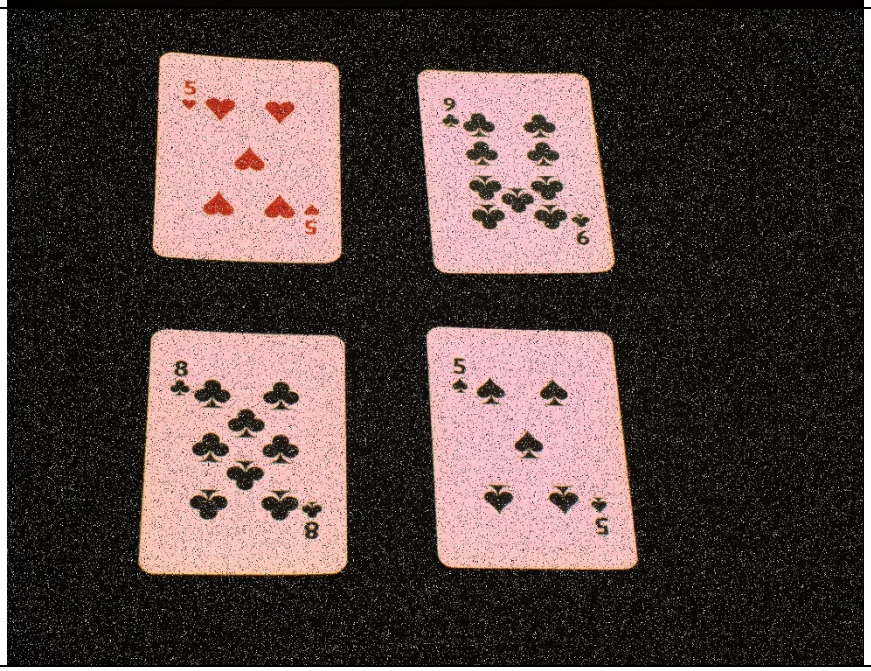
9.			3.png 8 Tile 6 Clover 6 Tile 8 Pike Harmonic mean = 6.85714
10.			3_blurred.png 8 Tile 6 Clover 6 Tile 8 Pike Harmonic mean = 6.85714

11.		3_gradient.png 8 Tile 6 Clover 6 Tile 8 Pike Harmonic mean = 6.85714
12.		3_noised.png 8 Tile 6 Clover 6 Tile 8 Pike Harmonic mean = 6.85714

13.			4.png 2 Clover 7 Pike 8 Tile 9 Heart Harmonic mean = 4.55079
14.			4_blurred.png 2 Clover 7 Pike 8 Tile 9 Heart Harmonic mean = 4.55079

15.		4_gradient.png 2 Clover 7 Pike 8 Tile 9 Heart Harmonic mean = 4.55079
16.		4_noised.png 2 Clover 7 Pike 8 Tile 9 Heart Harmonic mean = 4.55079

17.			5.png 8 Clover 5 Pike 9 Clover 5 Heart Harmonic mean = 6.28821
18.			5_blurred.png 8 Clover 5 Pike 9 Clover 5 Heart Harmonic mean = 6.28821

19.		5_gradient.png 8 Clover 5 Pike 9 Clover 5 Heart Harmonic mean = 6.28821
20.		5_noised.png 8 Clover 5 Pike 9 Clover 5 Heart Harmonic mean = 6.28821

Rzeczywisty efekt działania programu (dane wyjściowe) dla powyższych obrazów (danych wejściowych) wyglądają tak jak na rysunku poniżej:

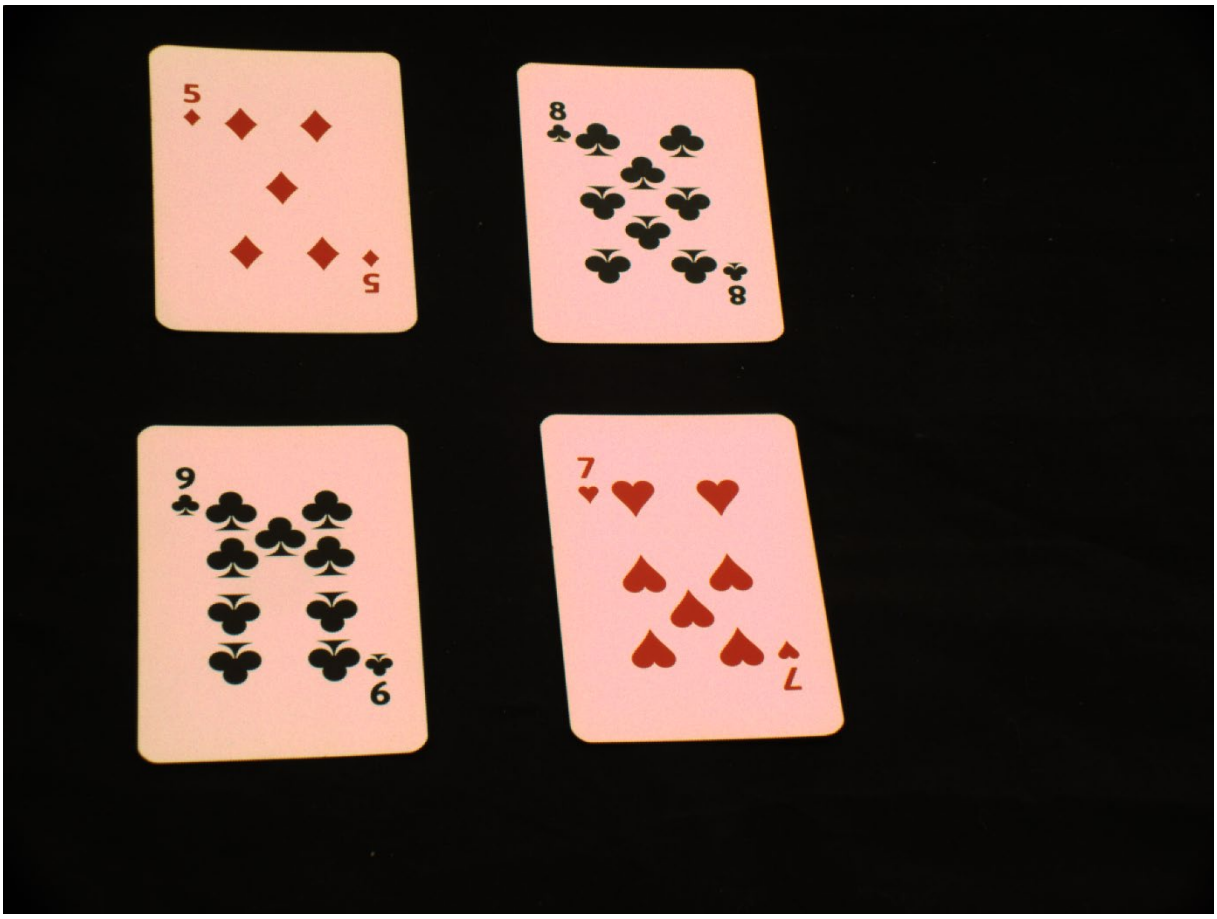
```
Number of images: 20
Number of cards: 80
1.png           9 Clover, 7 Heart, 8 Clover, 5 Tile,   Harmonic mean = 6.90884
1_blurred.png   9 Clover, 7 Heart, 8 Clover, 5 Tile,   Harmonic mean = 6.90884
1_gradient.png  9 Clover, 7 Heart, 8 Clover, 5 Tile,   Harmonic mean = 6.90884
1_noised.png    9 Clover, 7 Heart, 8 Clover, 5 Tile,   Harmonic mean = 6.90884
2.png           9 Tile, 10 Heart, 7 Clover, 5 Heart,   Harmonic mean = 7.22063
2_blurred.png   10 Heart, 9 Tile, 7 Clover, 5 Heart,   Harmonic mean = 7.22063
2_gradient.png  9 Tile, 10 Heart, 7 Clover, 5 Heart,   Harmonic mean = 7.22063
2_noised.png    9 Tile, 10 Heart, 7 Clover, 5 Heart,   Harmonic mean = 7.22063
3.png           8 Tile, 6 Clover, 6 Tile, 8 Pike,   Harmonic mean = 6.85714
3_blurred.png   8 Tile, 6 Clover, 6 Tile, 8 Pike,   Harmonic mean = 6.85714
3_gradient.png  8 Tile, 6 Clover, 6 Tile, 8 Pike,   Harmonic mean = 6.85714
3_noised.png    8 Tile, 6 Clover, 6 Tile, 8 Pike,   Harmonic mean = 6.85714
4.png           2 Clover, 7 Pike, 8 Tile, 9 Heart,   Harmonic mean = 4.55079
4_blurred.png   2 Clover, 7 Pike, 8 Tile, 9 Heart,   Harmonic mean = 4.55079
4_gradient.png  2 Clover, 7 Pike, 8 Tile, 9 Heart,   Harmonic mean = 4.55079
4_noised.png    2 Clover, 7 Pike, 8 Tile, 9 Heart,   Harmonic mean = 4.55079
5.png           8 Clover, 5 Pike, 9 Clover, 5 Heart,   Harmonic mean = 6.28821
5_blurred.png   8 Clover, 5 Pike, 9 Clover, 5 Heart,   Harmonic mean = 6.28821
5_gradient.png  8 Clover, 5 Pike, 9 Clover, 5 Heart,   Harmonic mean = 6.28821
5_noised.png    8 Clover, 5 Pike, 9 Clover, 5 Heart,   Harmonic mean = 6.28821
```

Rysunek 1

Widać, że program prawidłowo rozpoznaje wszystkie sceny i wszystkie warianty, różnice w kolejności opisywanych kart wynikają z kolejności w jakiej analizowane są piksele w obrazie.

4. Projekt rozwiązania

Jak już wyżej napisano, jedynym ostatecznym efektem działania programu jest wyświetlenie tekstowych informacji, jednakże do celów wykorzystania w niniejszej dokumentacji podczas pisania programu stworzono tymczasowe obiekty Mat umożliwiające podgląd wpływu poszczególnych operacji na obraz. Większość operacji będzie pokazana na przykładzie poniższego obrazu:



Rysunek 2

Poniższy podział na etapy pracy programu jest zgodny z wymaganiami przedstawionymi na zajęciach, jednak możliwe jest, że nie będzie on dokładnie pasował do rzeczywistego procesu przetwarzanie obrazu w programie.

- **Przetwarzanie wstępne**

Za przetwarzanie wstępne można uznać wczytanie wszystkich zdjęć do specjalnie stworzonej struktury `<img_data>`, która zawiera (oprócz każdego obrazu jako macierzy Mat) również nazwę danego obrazu, który z pomocą specjalnie napisanej funkcji jest uzyskiwany ze ścieżki pliku. Po tym etapie obrazy są przechowywane w strukturze wraz z ich nazwami.

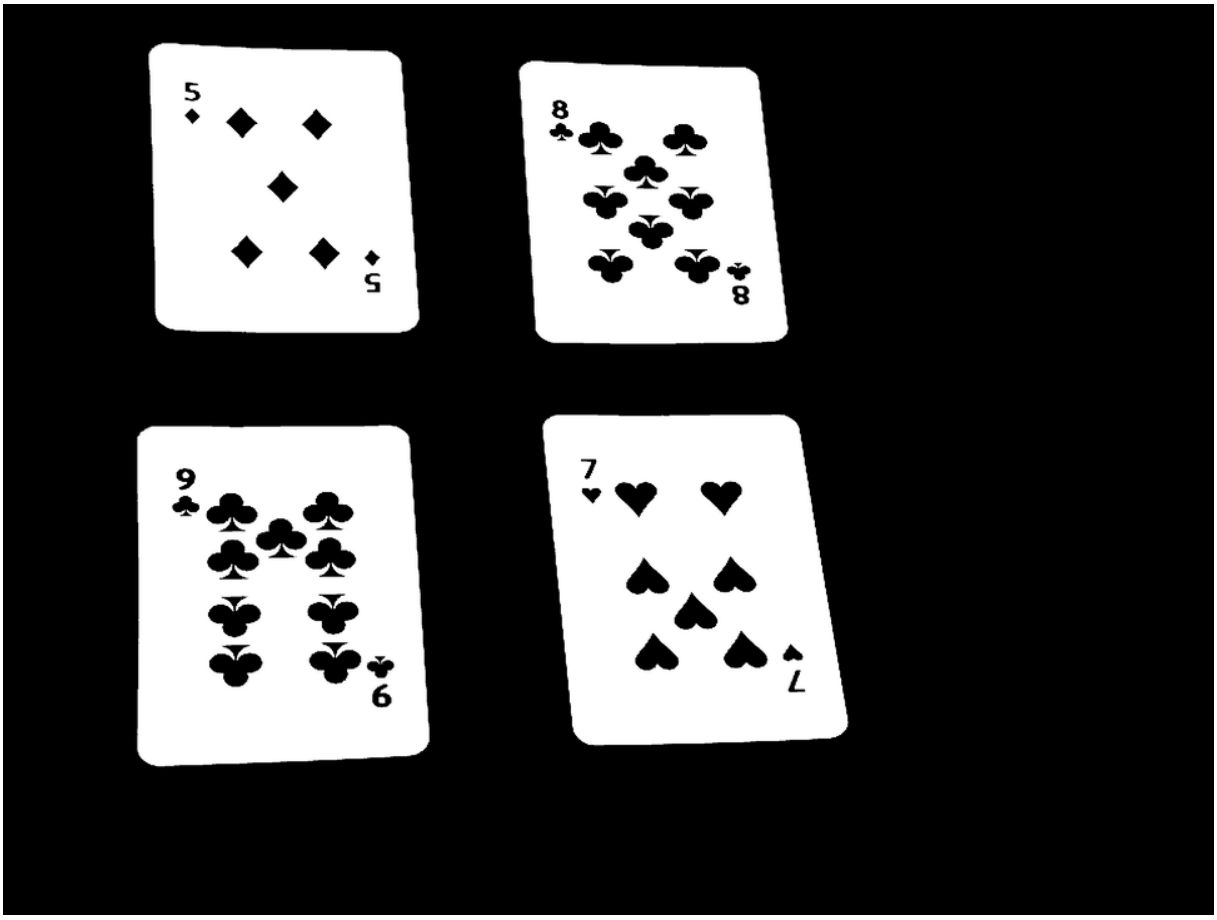
```
vector<String> images_paths()
{
	String path("Affek_karty//*.png");
	vector<String> pathes;
	glob(path, pathes, false);
	return pathes;
}
vector<img_data> load_source_img_data()
{
	vector<String> pathes = images_paths();
	vector<img_data> table_images(pathes.size());
	for (int i = 0; i < pathes.size(); i++)
	{
		Mat current_image = imread(pathes[i]);
		table_images[i].table_img = current_image;
		string name = pathes[i];
```

```
            size_t char_number = name.find_last_of('/');
            name = name.substr(char_number + 1);
            table_images[i].img_name = name;
        }
        return table_images;
}
```

- **Uwydatnienie cech obiekt tło**

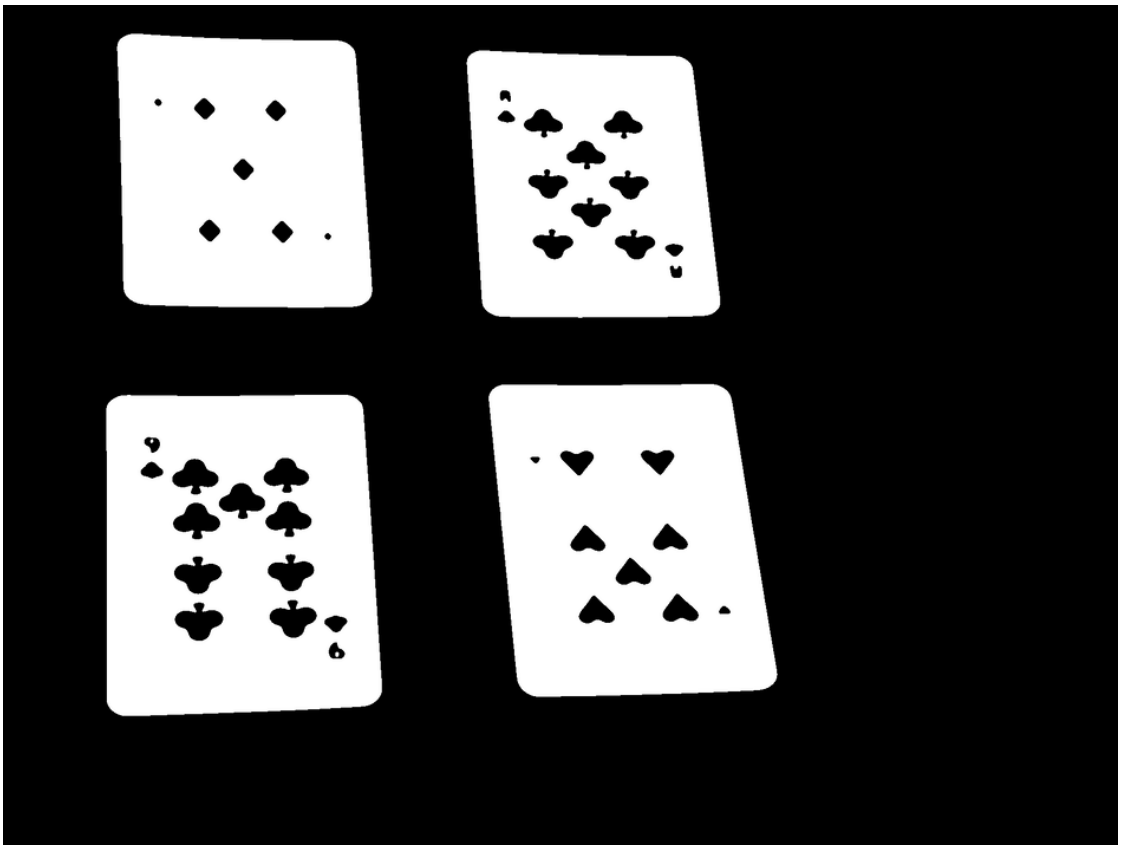
Uwydatnianie cechy obiekt – tło następuje poprzez progowanie i binaryzację. Białe karty umieszczone są na czarnym tle, więc progowanie nie jest problemem. Zastosowany został algorytm progowania Otsu, jednakże w tym przypadku nawet najprostszy algorytm z podanym odgórnie progiem np. 127 byłby skuteczny. Kod programu oraz efekt działania progowania widoczny jest poniżej:

```
Mat thresholding(Mat source_img, int min_thresh, int max_thresh)
{
        Mat output_img, gray_img;
        cvtColor(source_img, gray_img, COLOR_BGR2GRAY);
        threshold(gray_img, output_img, min_thresh, max_thresh, THRESH_BINARY |
CV_THRESH_OTSU);
        return output_img;
}
```

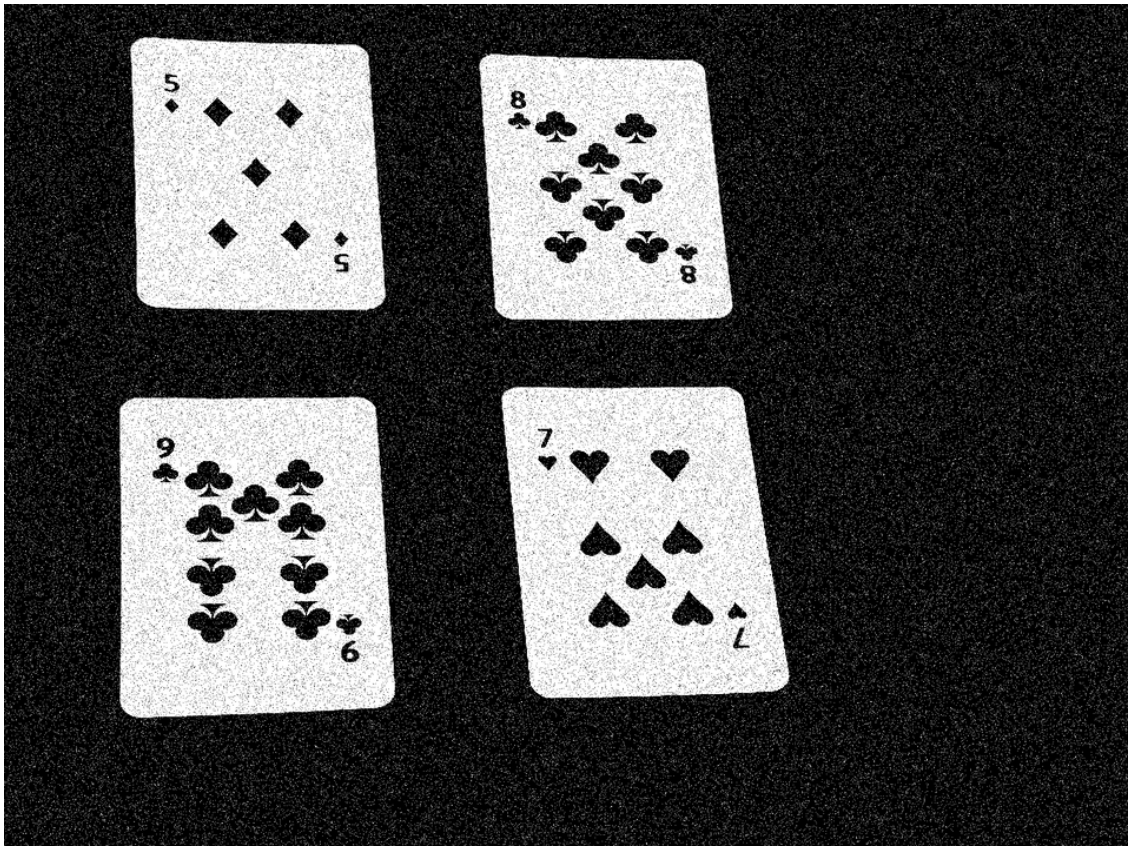


Rysunek 3

Dla porównania poniżej ten sam efekt dla obrazu rozmytego oraz zaszumionego:



Rysunek 4



Rysunek 5

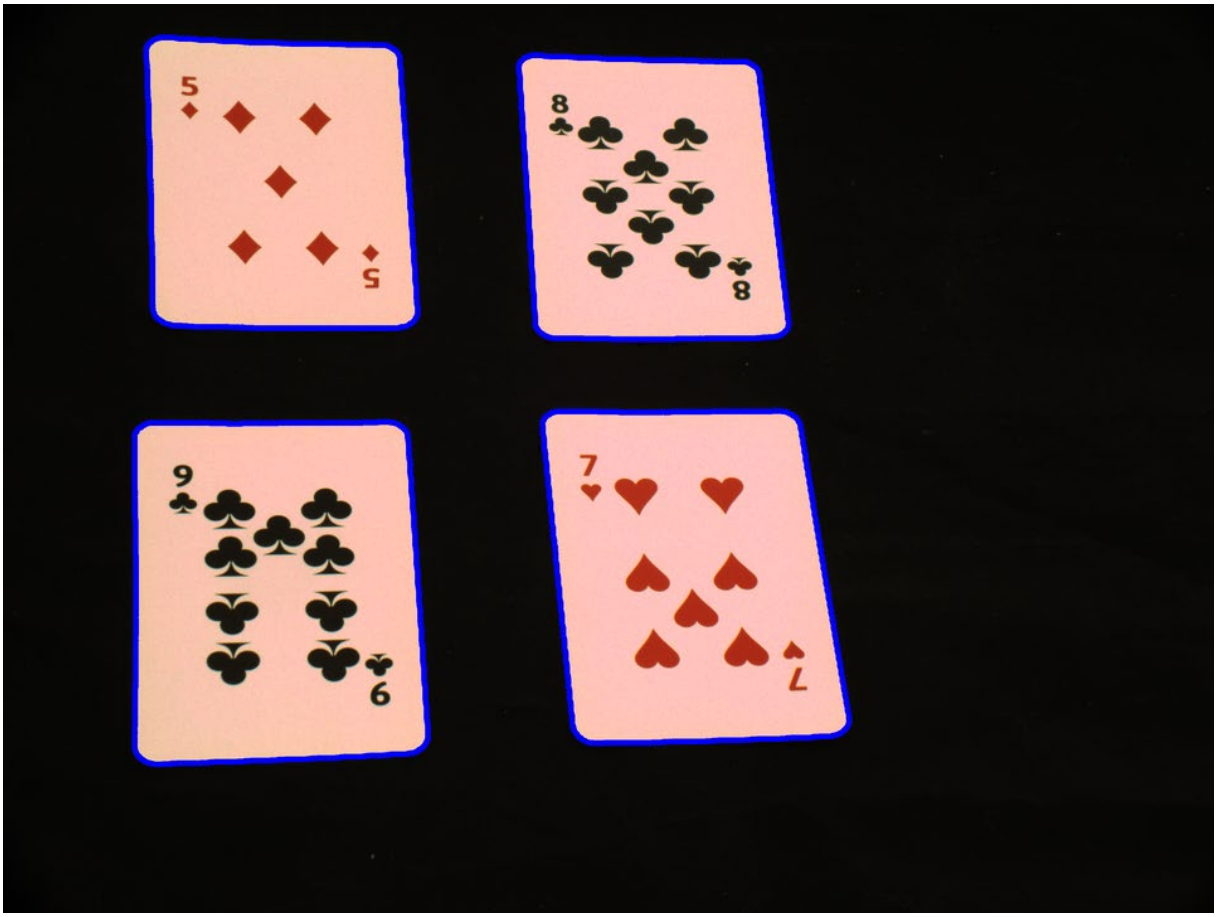
- **Filtracja**

Na tym etapie nie ma potrzeby stosowania dodatkowych filtrów, ponieważ po binaryzacji posegmentowanie kart nie stanowi problemu. Dodatkowa filtracja obrazów odbywa się w następnych krokach, po segmentacji kart, więc będzie omówiona w kolejnych punktach.

- **Segmentacja kart**

Segmentacja kart polega na wykorzystaniu funkcji findContours, która wyszukuje wszystkie kontury w obrazie. Aby wydzielić kontury jedynie 4 kart w obrazie wystarczy znaleźć 4 największe kontury w zbinaryzowanym obrazie. Odbywa się to poprzez zastosowanie warunku powierzchni konturu (większy niż określona wartość). Efekt działania tej operacji przedstawia poniższy rysunek. Dla przejrzystości wyznaczone kontury zostały narysowane na obrazie źródłowym, jednak ich wyznaczanie opiera się na danych z obrazu binarnego. Użyto do tego funkcji drawContours. Dla każdego z wariantów (gradient, blur, szum) operacja znajdowania konturów kart działa równie dobrze.

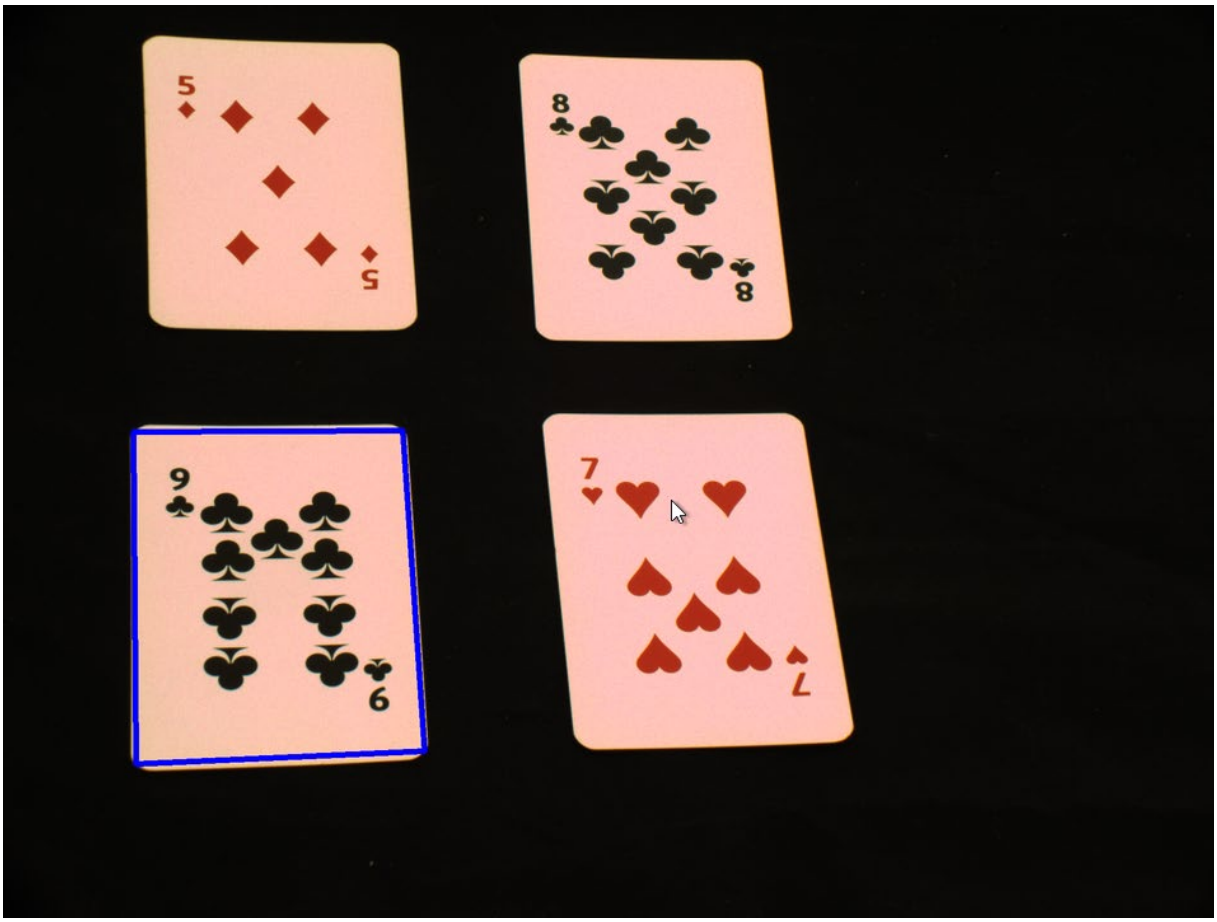
```
findContours(thresholded, all_img_contours, contours_hierarchy, CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
```



Rysunek 6

- **Przetwarzanie dodatkowe obszarów kart**

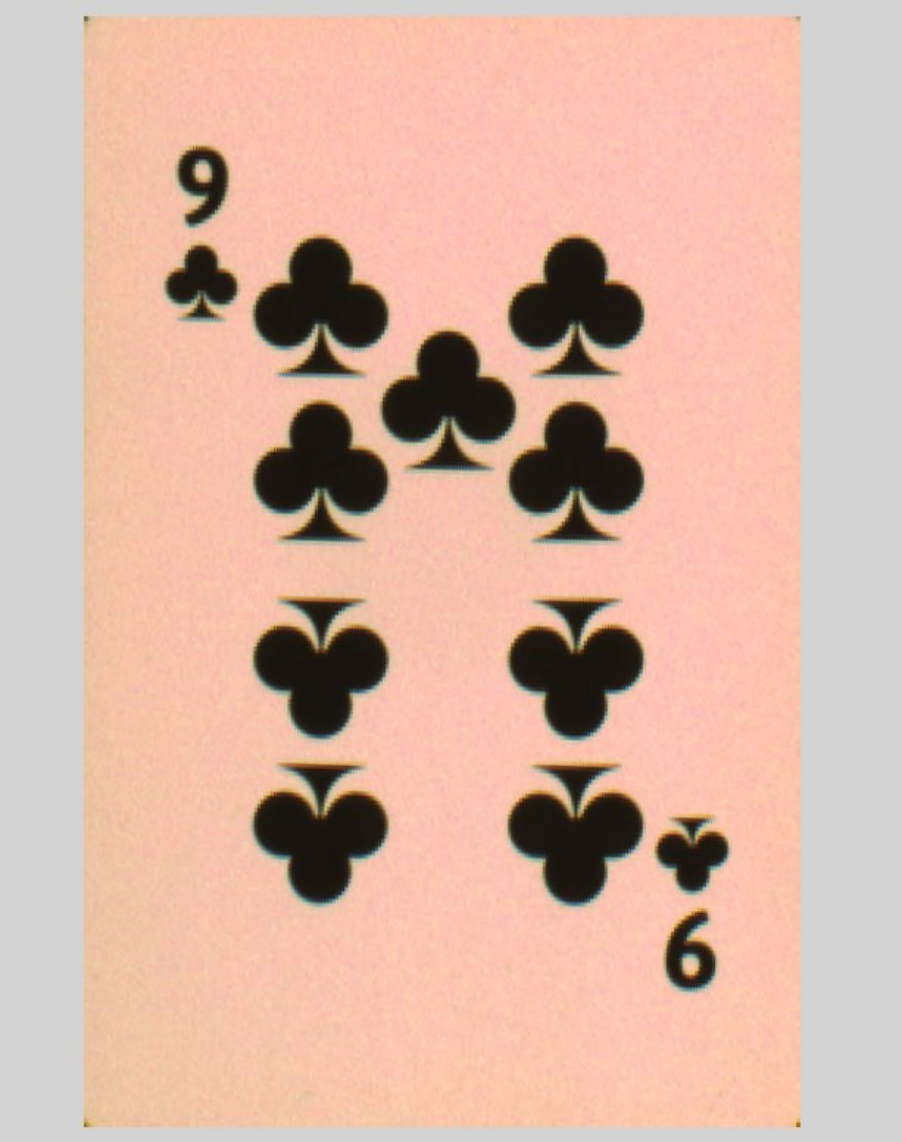
Zanim przystąpi się do pracy z wykrywaniem symboli należało w pewnym stopniu ustandaryzować kształt i wymiary symboli na kartach. Możliwe jest to poprzez np. poprawienie perspektywy, czyli doprowadzenie do tego samego wymiaru każdej karty. Mając wykryte kontury kart można taki kontur przybliżyć wielokątem (w przypadku karty czterokątem). Biblioteka OpenCV posiada funkcję dedykowaną do takich zadań - `approxPolyDP();` Umożliwia ona określenie jak duże to będzie przybliżenie (jaka może być największa odległość między konturem a jego przybliżeniem), dopasowując odpowiednio wartości, zawsze jest możliwe przybliżenie karty czterokątem. Przybliżenie dla jednej z kart przedstawia rysunek poniżej:



Rysunek 7

Kolejnym krokiem jest wykorzystanie funkcji `getPerspectiveTransform();` oraz `warpPerspective();` Pierwsza z nich wyznacza macierz przekształcenia (pobierając współrzędne punktów przybliżonego konturu oraz punktów kształtu, który chcemy uzyskać – w tym przypadku prostokąta). Druga funkcja dokonuje samego przekształcenia korzystając z obrazu wejściowego i macierzy przekształcenia. Aby transformacja się udała to należy wiedzieć, które rogi przybliżonego wielokąta należy przetransformować na rogi prostokąta, a ponieważ funkcja wyszukująca kontury analizuje obraz zgodnie z pewną logiką, to zależnie od ułożenia karty (np. trochę ukośnie) nie zawsze udaje się otrzymać identyczną kolejność punktów w wektorze, należało więc stworzyć funkcję sortującą rogi czworokąta poczynając

od lewego górnego rogu, odwrotnie do wskazówek zegara. Efekt dla tej samej karty co powyżej widoczny jest na poniższym rysunku:



Rysunek 8

Powyższe operacje realizowane są przez przedstawione poniżej funkcje:

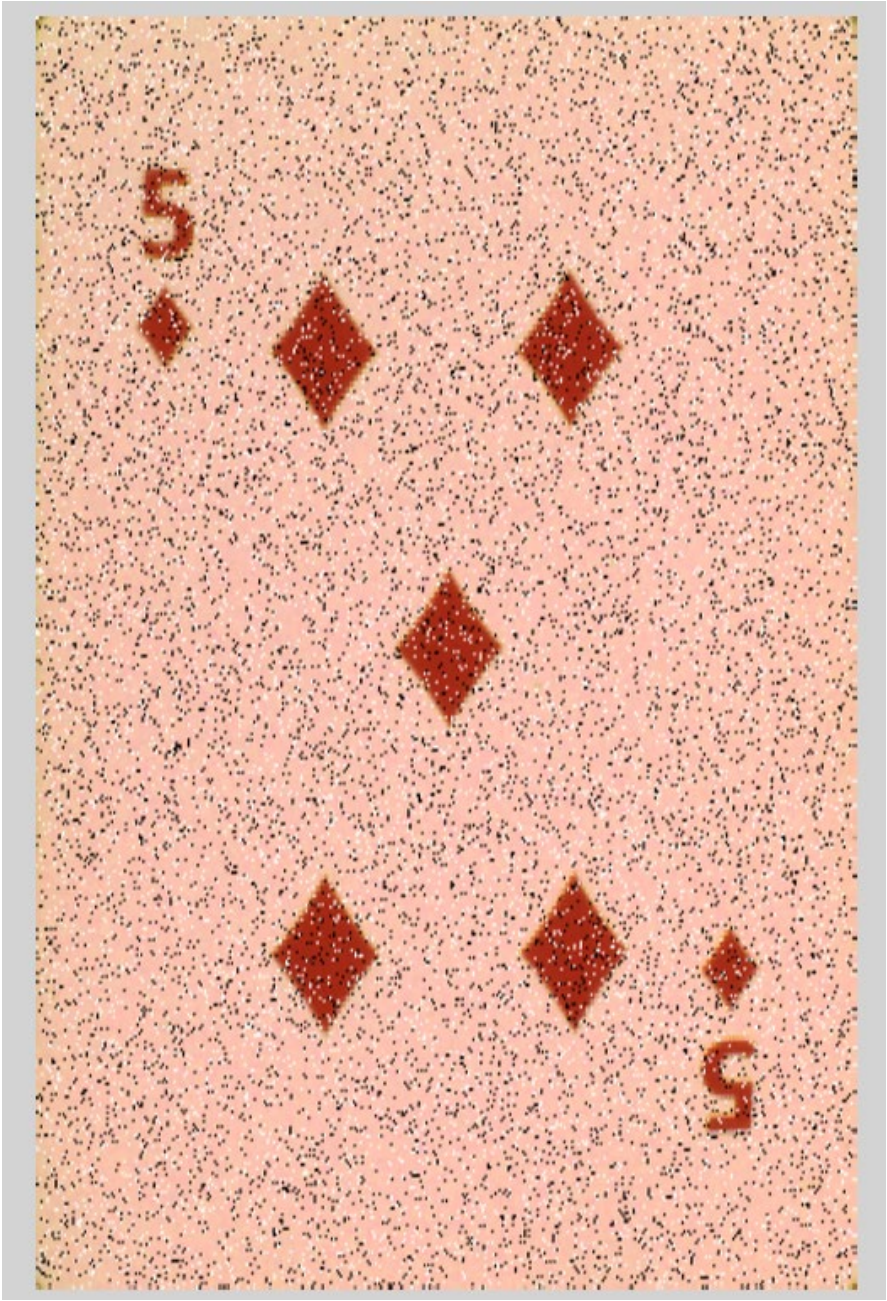
```
void good_perspective(vector<img_data> &cards_images, const Point2f sorted_corners[],
int card_num, int image_num)
{
	Point2f output_cards[] = { Point2f(0,0), Point2f(0, 700), Point2f(450, 700),
Point2f(450, 0) };
	Mat lambda = getPerspectiveTransform(sorted_corners, output_cards);
	warpPerspective(cards_images[image_num].table_img,
cards_images[image_num].card_img[card_num], lambda, Size(450, 700));
}
void cards_corners_sorted(vector<img_data> &cards_images, const vector<vector<Point>>
finded_contours, int card_num, int contour_num, int image_num)
{
	vector<Point> approx(4);
	Point2f approx_sorted[4];
	approxPolyDP(finded_contours[contour_num], approx, 40, true);
	int point_index = 0;
```

```
	for (int i = 0; i < 3; i++)
		if (approx[i + 1].y + approx[i + 1].x < approx[point_index].y +
approx[point_index].x)
			point_index = i + 1;
	for (int i = 0; i < 4; i++)
		approx_sorted[i] = approx[(point_index + i) % 4];
	good_perspective(cards_images, approx_sorted, card_num, image_num);
}
```

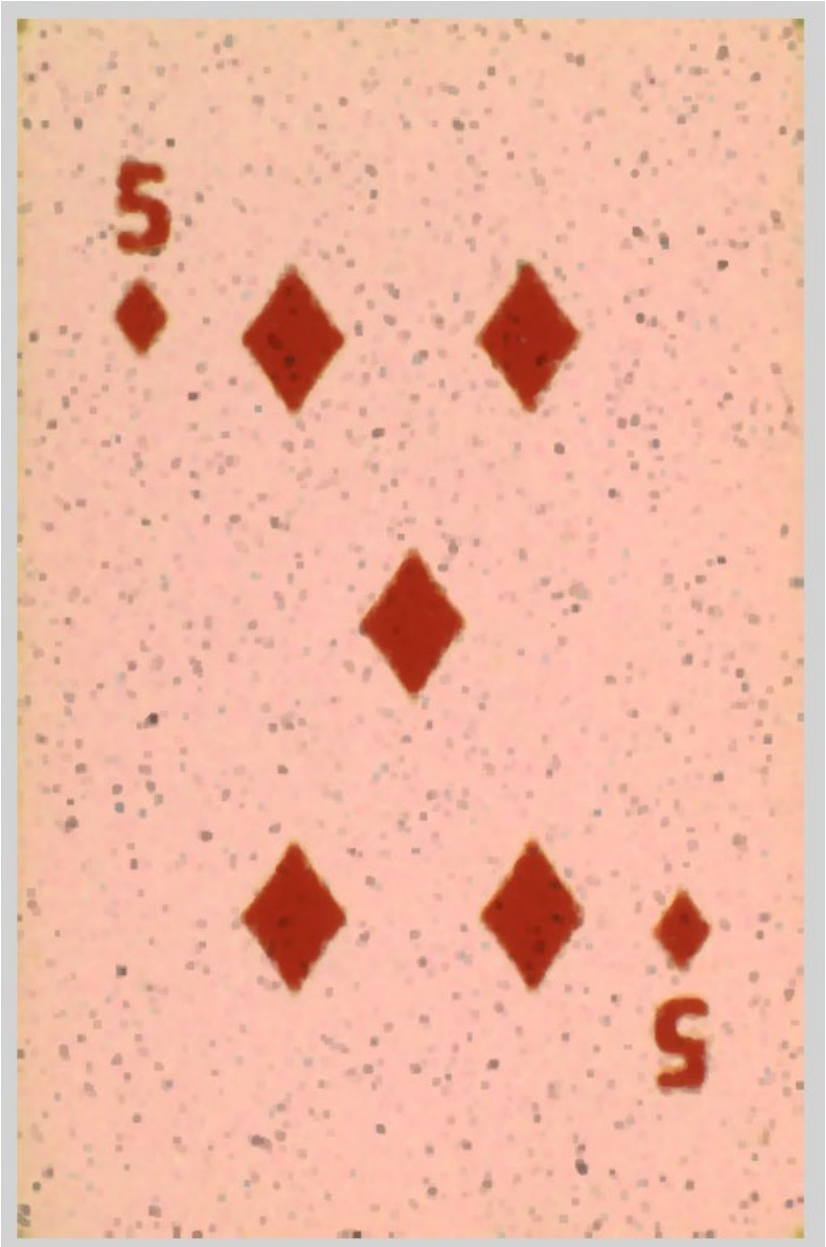
Ze względu na to, że w dalszej części programu nie będą już potrzebne oryginalne obrazy z 4 kartami to wykryte karty są przepisywane do nowej struktury (umożliwiającej przechowywanie nazw plików, pojedynczych kart, zbinaryzowanych kart, rodzajów rozpoznanych kart i ich wartości oraz średnią harmoniczną). Następnie poprzednia struktura jest „czyszczona” z danych co zwalnia większą część pamięci operacyjnej:

```
vector<img_data>().swap(table_images_data);
```

Mając już przygotowane obrazy można skorzystać z kilku filtrów aby usunąć niechciany wpływ m.in. szumu na wyniki. Zestaw filtrów należało dobrać tak, aby można było każdy z nich wykorzystać na wszystkich zdjęciach, oznaczało to, że filtr niwelujący szum nie mógł jednocześnie „popsuć” zdjęcia oryginalnego – ostrego.
Pierwszym wybranym filtrem, który zastosowano na wszystkie karty w równym stopniu był filtr medianowy – idealnie nadaje się on do niwelowania szumu typu „sól i pieprz” – jednocześnie nie zdegradował on jakości pozostałych zdjęć – efekt działania filtra widać na poniższych rysunkach:



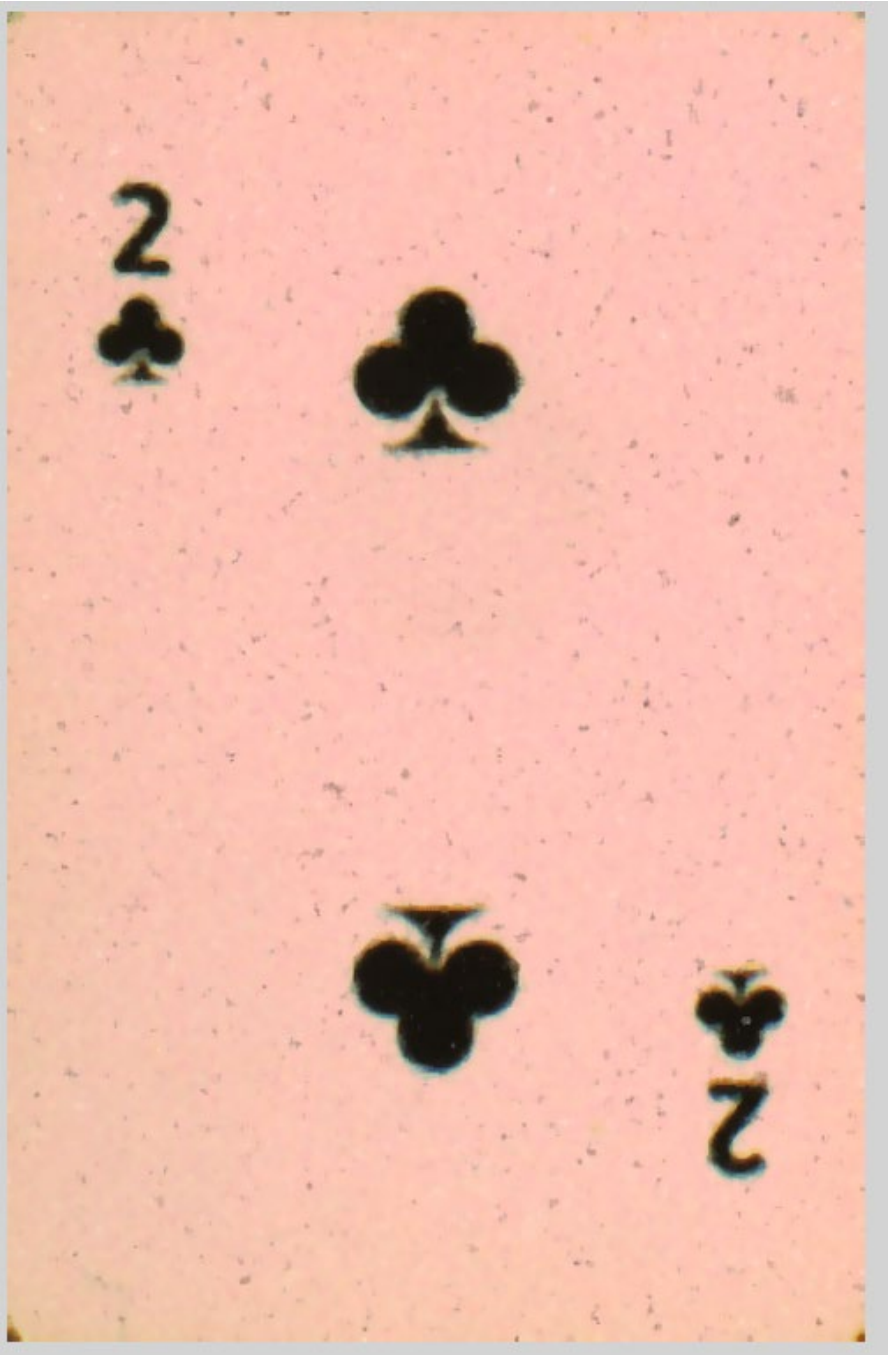
Rysunek 9



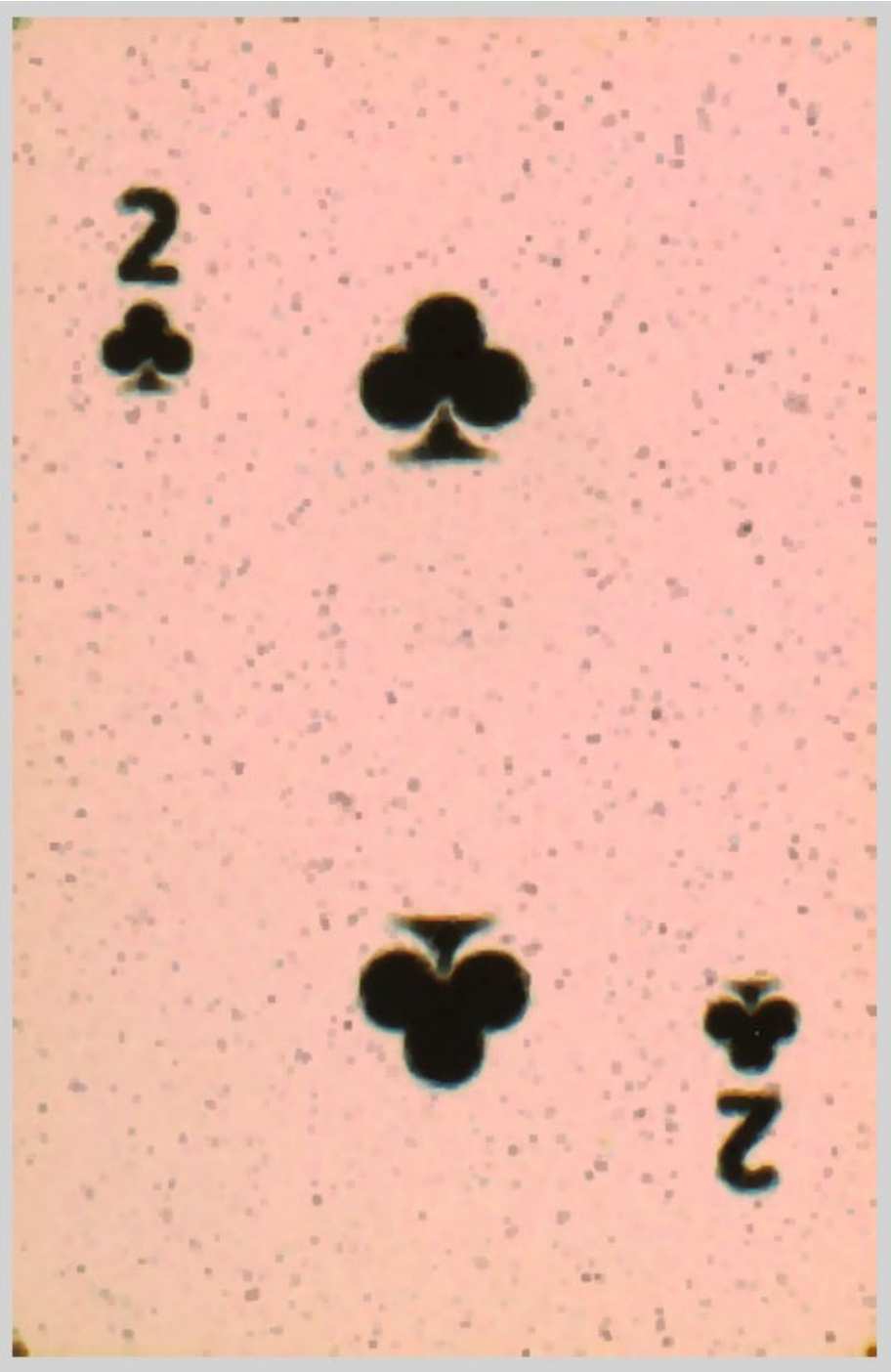
Rysunek 10

Jak widać zdecydowana większość szumu została usunięta.

Kolejnym filtrem koniecznym do zastosowania przed segmentacją jest operacja morfologiczna erozji, efekt jej działania nie jest bardzo wyraźny gołym okiem, ale jest to istotne zwłaszcza w przypadku kart rozmytych oraz zaszumionych z symbolami takimi jak trefl lub pik, ponieważ ze względu na bardzo cienkie połączenie „ogonka” z resztą symbolu program podczas progowania lub wyszukiwania konturów mógłby potraktować to jako osobne znaki. Zastosowanie erozji spowoduje, że ciemne znaki się nieco „rozszerzą” i ich połączenie będzie bardziej widoczne. Efekt działania erozji dla obrazu zaszumionego widoczny jest poniżej:



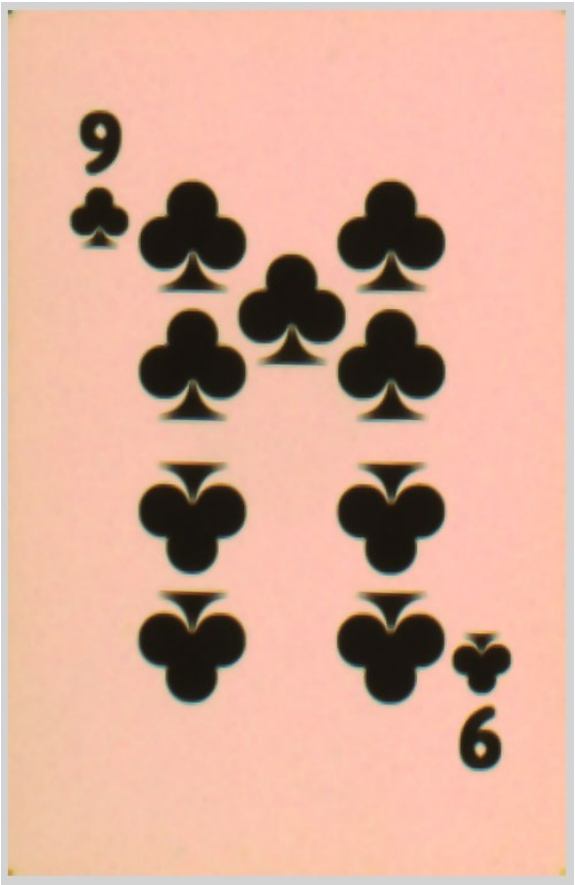
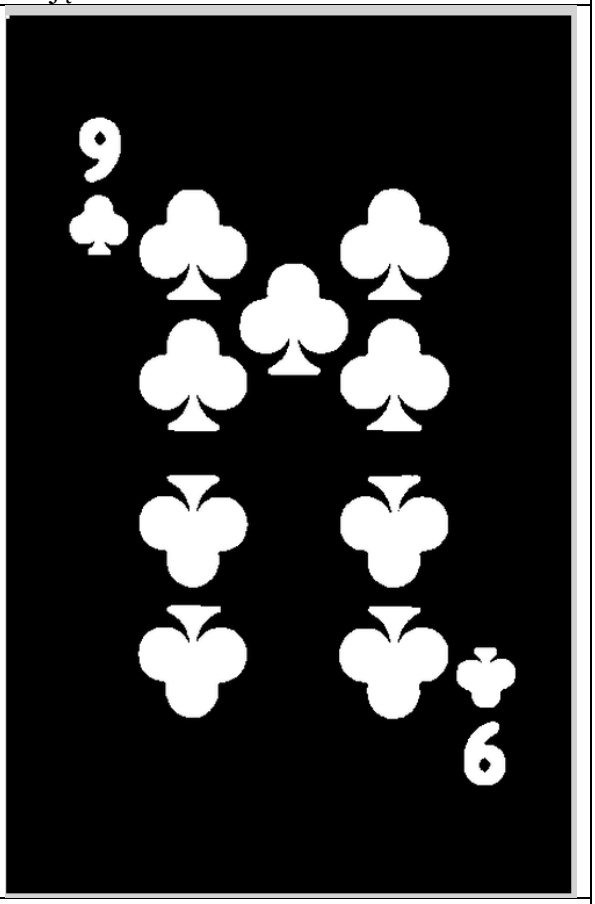
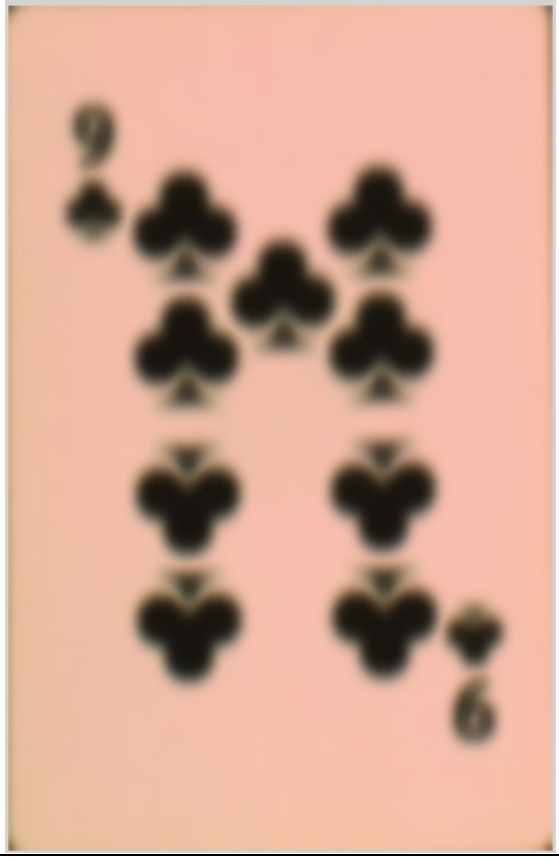
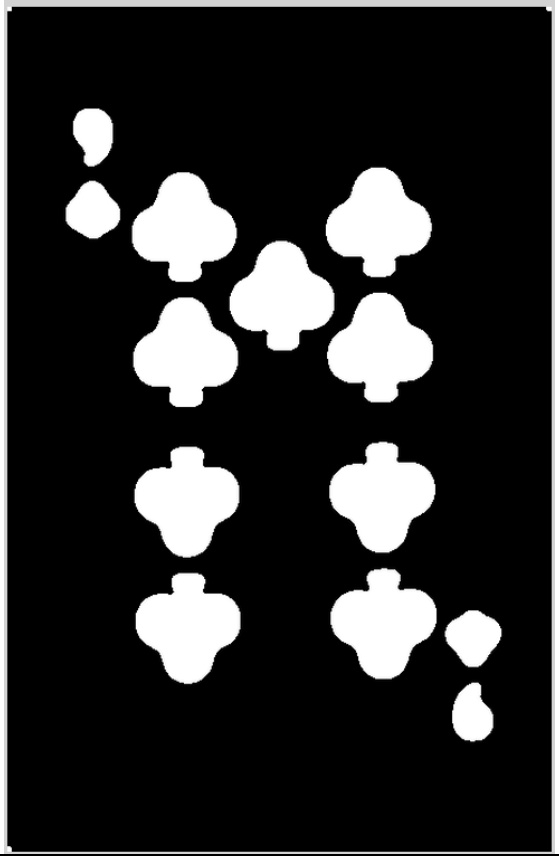
Rysunek 11

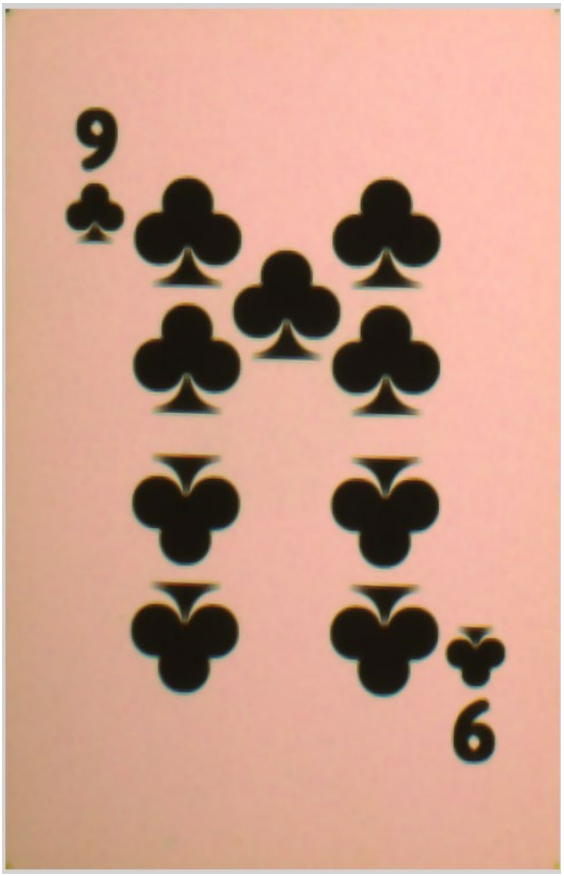
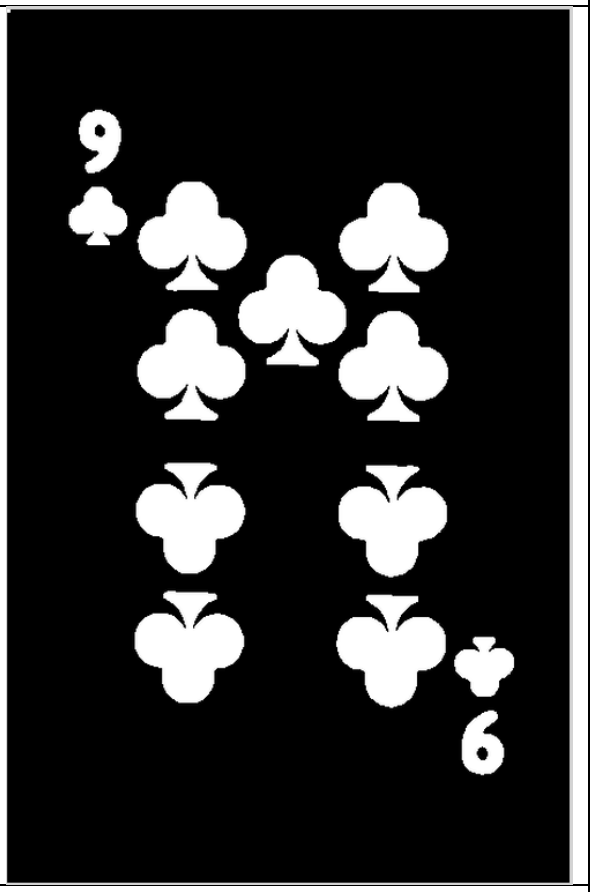
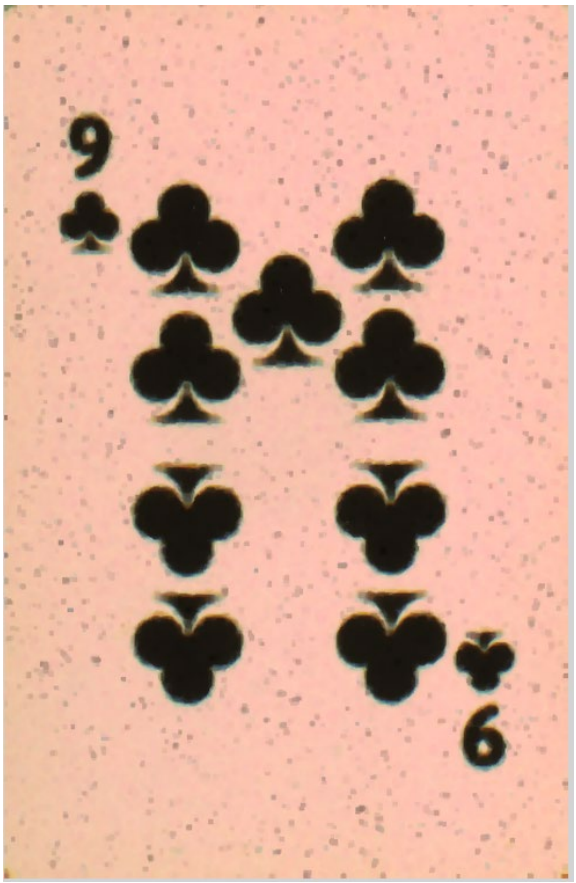
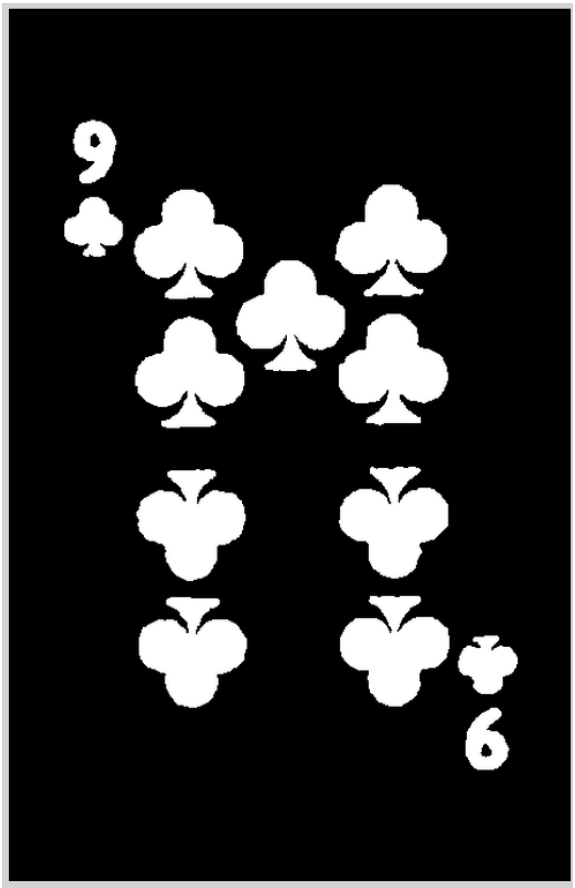


Rysunek 12

Jak widać oprócz zmniejszenia się (lub nawet zaniku) odległości części głównej symbolu i odstającej „nóżki” zwiększyła się również wielkość drobinek szumu. Nie jest to jednak problem, nie przeszkadza to w dalszej części procesu.

Ostatnimi operacjami przed segmentacją symboli jest progowanie i binaryzacja metodą Otsu oraz inwersja kolorów, ponieważ do wyszukiwania konturów szukane obiekty muszą być białe. Poniższa tabelka przedstawia tę samą kartę dla 4 różnych wariantów po i przed tymi dwiema operacjami.

Przed	Po
Oryginalne zdjęcie	
	
Obraz rozmyty	
	

Obraz z gradientem	
	
Obraz zaszumiony	
	

Ze względu na to, że każdy z wymienionych powyżej filtrów jest realizowany jedną linijką w bibliotece OpenCV, więc zostały one umieszczone w jednej funkcji w celu uproszczenia kodu:

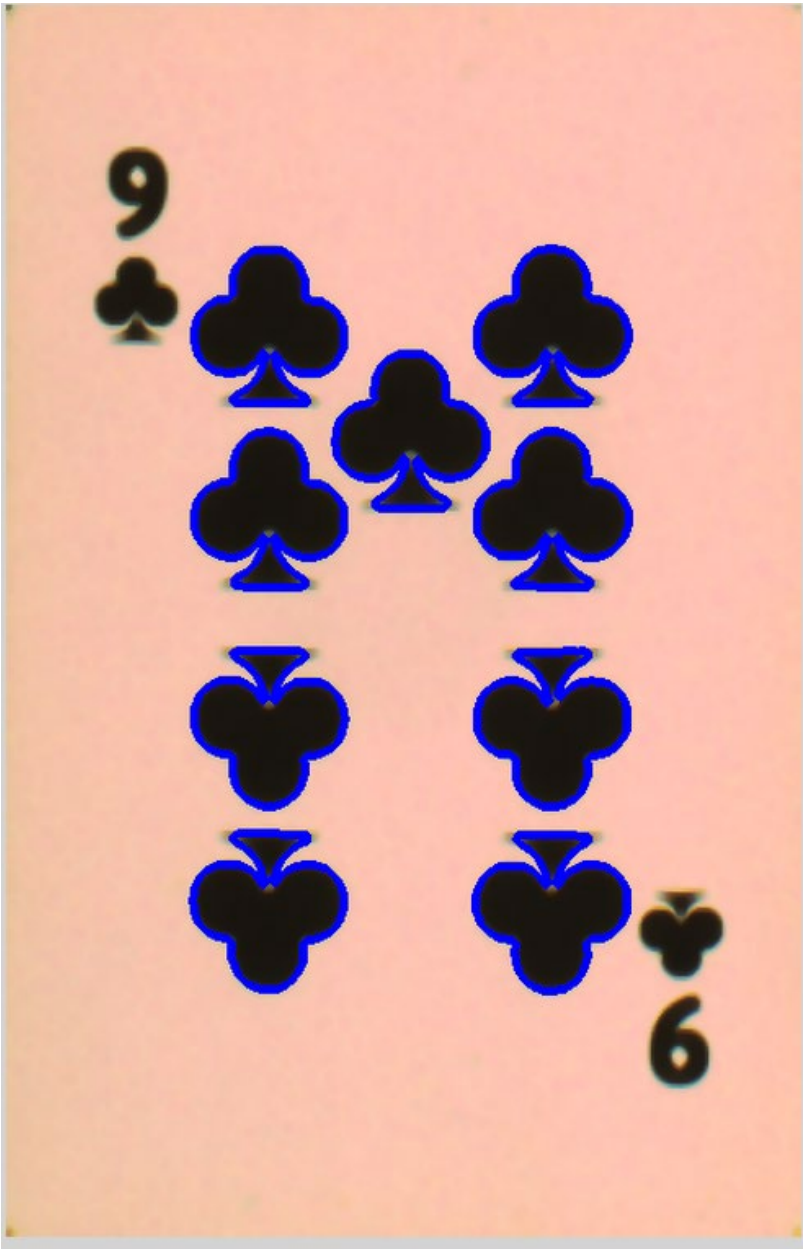
```
void base_filters(vector<single_cards> &detected_cards, int table_img_num, int
card_num)
{
	medianBlur(detected_cards[table_img_num].card[card_num],
detected_cards[table_img_num].card[card_num], 5);
	Mat element = getStructuringElement(MORPH_RECT, Size(3, 3), Point(-1, -1));
	erode(detected_cards[table_img_num].card[card_num],
detected_cards[table_img_num].card[card_num], element);
	detected_cards[table_img_num].thresholded_card[card_num] =
thresholding(detected_cards[table_img_num].card[card_num], 150, 255);
	bitwise_not(detected_cards[table_img_num].thresholded_card[card_num],
detected_cards[table_img_num].thresholded_card[card_num]);
}
```

- **Segmentacja symboli**

Segmentacja symboli jest realizowana dzięki funkcji szukania konturów, dodatkowym warunkiem jest tutaj minimalna powierzchnia konturu, która jest określona w taki sposób aby wszystkie główne symbole były wykrywane, natomiast poboczne w rogach kart – pomijane. Efekt takiego działania przedstawia poniższy rysunek, oczywiście szukanie konturów odbywa się na obrazie binarnym, jednak w celu zwiększenia przejrzystości zaznaczone one zostały na obrazie wejściowym. Dzięki temu wartość danej karty jest już trywialna do policzenia, wystarczy policzyć zaznaczone kontury na obrazie.

Kod:

```
drawContours(detected_cards[k].original[i], contours, j, Scalar(255, 0, 0), 4, 8,
hierarchy);
```

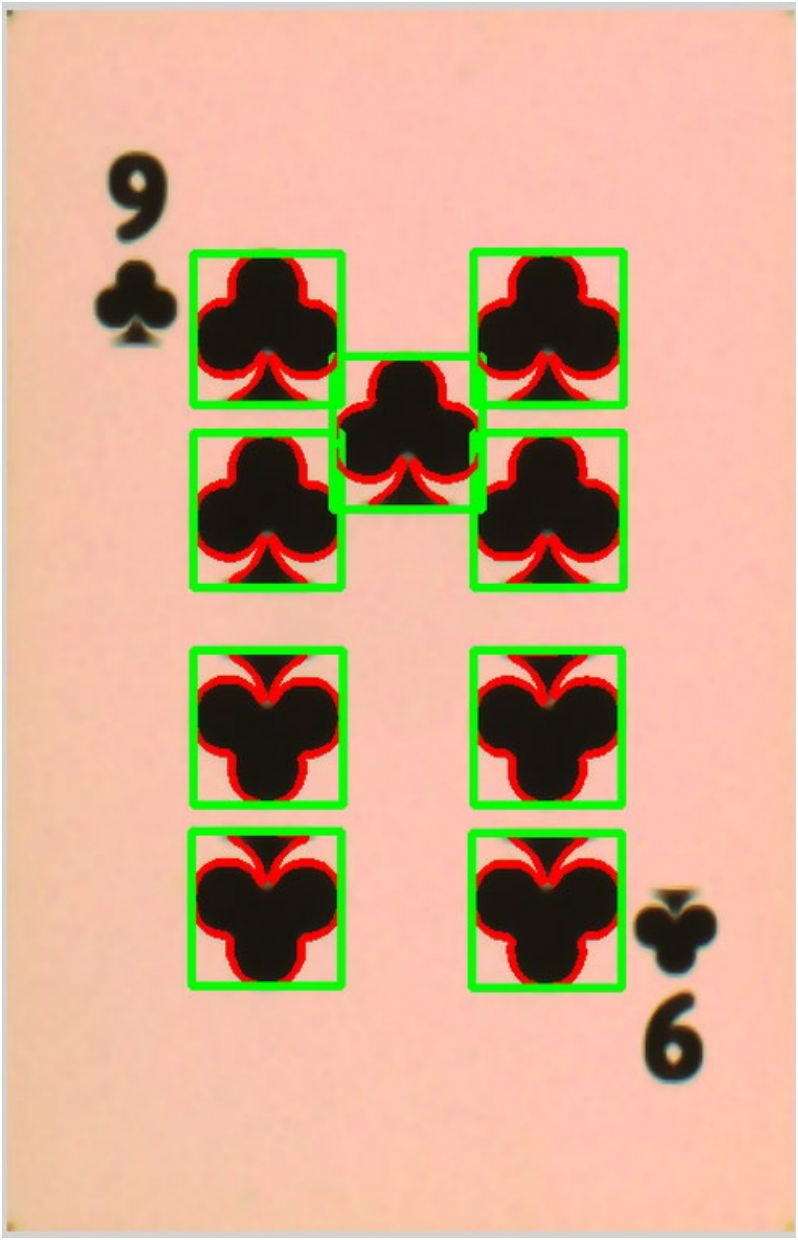


Rysunek 13

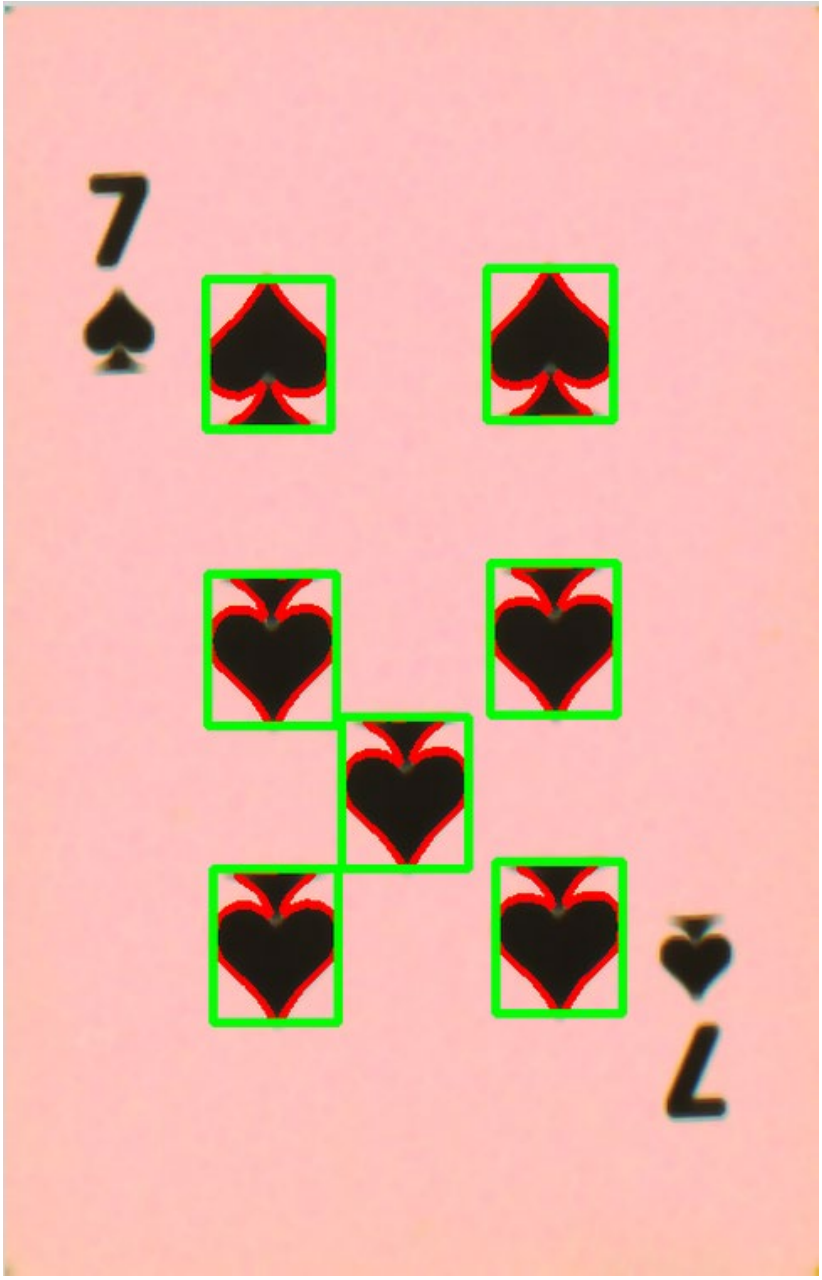
- **Obliczenie cech**

Po analizie postawionego zadania, okazało się, że parą cech, które wystarczają do jednoznacznego określenia symbolu karty (dla tak nieznacznie różniących się obrazów) są barwa symbolu oraz stosunek boków prostokąta opisanego na konturze. Podział na barwy jest oczywisty, natomiast w przypadku czarnych symboli prostokąt opisany na treflu jest bardzo zbliżony do kwadratu, czyli stosunek jego boków wynosi ok. 1, natomiast w przypadku symbolu pik stosunek szerokości do wysokości jest zawsze mniejszy od 0,9. W przypadku symboli czerwonych stosunek szerokości do wysokości prostokąta opisanego na danym symbolu wynosi zawsze: dla kier >0,8, a dla karo <0,8. OpenCV oferuje specjalną funkcję umożliwiającą wyznaczenie prostokąta na konturze (opisanie): `boundingRect(contour);`

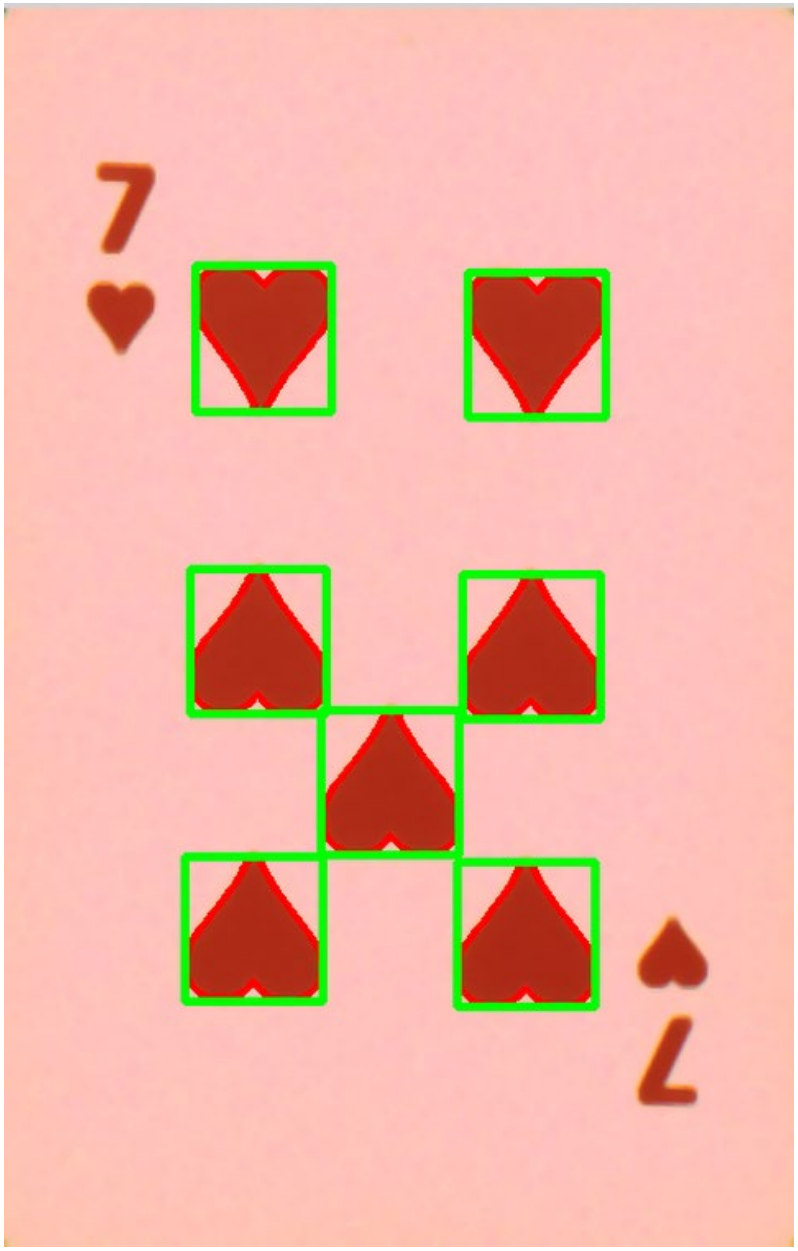
Różnice w stosunku szerokości do wysokości dla różnych symboli ilustrują poniższe 4 rysunki:



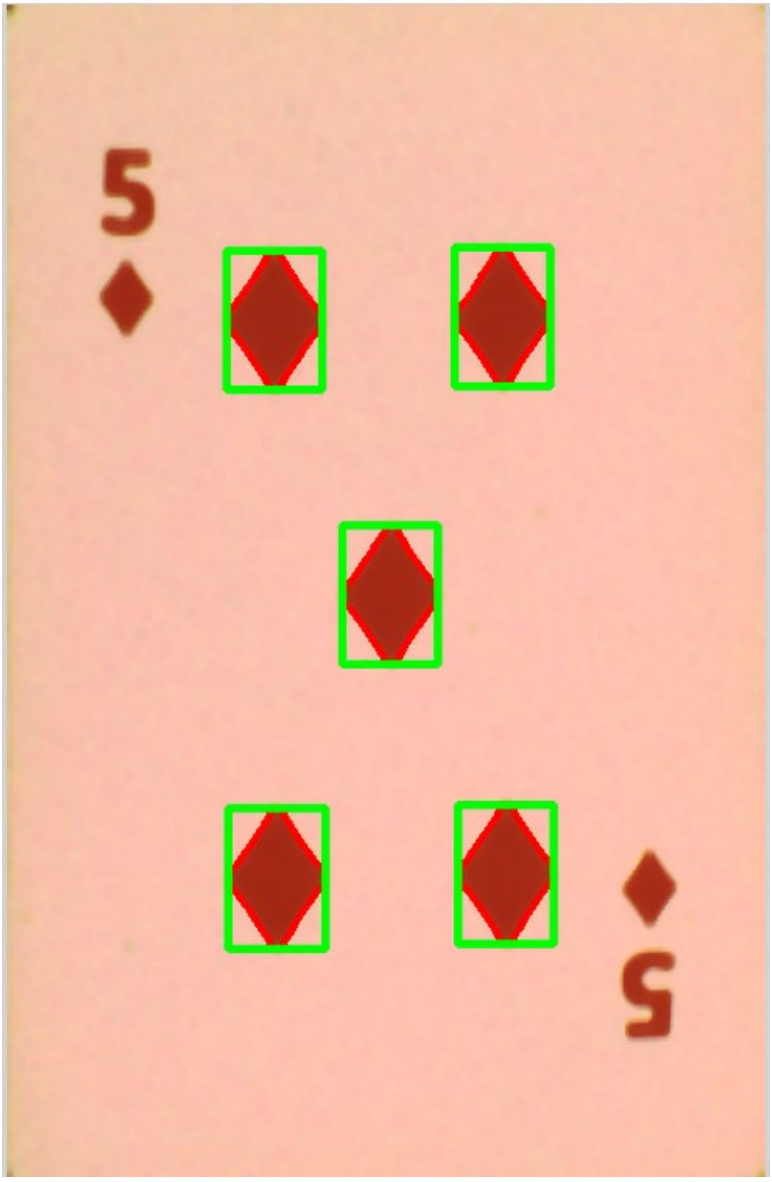
Rysunek 14 Trefl



Rysunek 15 Pik



Rysunek 16 Kier



Rysunek 17 Karo

Jak widać dla poszczególnych barw symboli opisane na nich prostokąty różnią się znacząco. Poniżej znajduje się kod – do obliczenia średniej barwy danego symbolu wykorzystano funkcję `mean(img, mask);` która umożliwia obliczenie średniej barwy w danym obszarze (wykorzystując kontur jako maskę). Na potrzeby projektu interesujący jest tylko kanał R – jeżeli jego średnia wartość wynosi poniżej 100 to mamy do czynienia z czarnymi symbolami, jeżeli powyżej to z czerwonymi.

```
int mean_symbols_color(const Mat single_card, const vector<vector<Point>> contours,
int contour_num, vector<Vec4i> hierarchy)
{
	Mat mask = Mat::zeros(701, 451, CV_8U);
	drawContours(mask, contours, contour_num, Scalar(255, 255, 255), -1, 8,
hierarchy);
	Scalar card_symbol_color = mean(single_card, mask);
	return card_symbol_color.val[2];
}
```

- **Klasyfikacja obiektów**

Mając dane z poprzedniego punktu można jednoznacznie określić każdą kartę. Poniższy kod przedstawia realizację klasyfikacji symbolu:

```
string symbols_recognition(const Mat single_card, const vector<vector<Point>>
contours, int contour_num, vector<Vec4i> hierarchy)
{
	string suit;
	int card_color = mean_symbols_color(single_card, contours, contour_num,
hierarchy);
	Rect symbol = boundingRect(contours[contour_num]);
	if ((float)symbol.width / symbol.height > 0.9 && card_color < 100)
		return "Clover";
	if ((float)symbol.width / symbol.height > 0.8 && card_color > 100)
		return "Heart";
	if ((float)symbol.width / symbol.height <= 0.9 && card_color < 100)
		return "Pike";
	if ((float)symbol.width / symbol.height <= 0.8 && card_color > 100)
		return "Tile";
}
```

Ostatnim etapem jest zapisanie wszystkich rozpoznanych informacji w strukturze:

```
struct single_cards
{
	String table_img_name;
	string suit[4];
	int value[4];
	Mat card[4];
	Mat thresholded_card[4];
	float harmonic_mean;
};
```

- **Realizacja części a) polecenia**

Pierwszą częścią polecenia było rozpoznanie kart znajdujących się na obrazach. W powyższych punktach zostało pokazane jak to zrobić oraz udowodnione, że metoda ta działa na wszystkich obrazach oraz wariantach, bez konieczności dostosowywania parametrów w zależności od wariantu. Dzięki zastosowaniu struktury, można się dostać do tych danych w każdej chwili pracy programu jak również wykonanie części drugiej polecenia nie jest już problemem. Ostatnim etapem działania programu jest wypisanie wszystkich wyliczonych informacji w konsoli. Kod poniżej:

```
void display_in_console(const vector<single_cards> detected_cards)
{
	cout << "Number of images: " << detected_cards.size() << endl;
	cout << "Number of cards: " << detected_cards.size() * 4 << endl;
	for (int i = 0; i < detected_cards.size(); i++)
	{
		cout << detected_cards[i].table_img_name << "\t";
		if (detected_cards[i].table_img_name.length() < 9)
			cout << "\t";
		for (int j = 0; j < 4; j++)
		{
			cout << detected_cards[i].value[j] << " " <<
detected_cards[i].suit[j] << ", ";
		}
		cout << "  Harmonic mean = " << detected_cards[i].harmonic_mean << endl;
	}
```

```
}
```

- **Realizacja części b) polecenia**

Część b) polecenia polegała na wyznaczeniu średniej harmonicznej wszystkich symboli na zdjęciu. Co to jest średnia harmoniczna można przeczytać w punkcie opis teoretyczny. Efekty obliczenia średniej harmonicznej można zobaczyć na pierwszych stronach dokumentacji. Kod wykorzystany do obliczenia znajduje się poniżej:

```
void harmonic_mean_calculation(vector<single_cards> &detected_cards)
{
	for (int i = 0; i < detected_cards.size(); i++)
	{
		float harmonic_mean = 0;
		harmonic_mean = 4.0 / ((1.0 / detected_cards[i].value[0]) + (1.0 /
detected_cards[i].value[1]) + (1.0 / detected_cards[i].value[2]) + (1.0 /
detected_cards[i].value[3]));
		detected_cards[i].harmonic_mean = harmonic_mean;
	}
}
```

5. Implementacja

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

Każda funkcja napisana w programie ma konkretne zadanie i wywołanie jej ma zawsze na celu wykonanie jednej i tej samej czynności, jednakże zdarza się, że funkcje takie jak np.

```
void base_filters(vector<single_cards> &detected_cards, int table_img_num, int
card_num)
{
	medianBlur(detected_cards[table_img_num].card[card_num],
detected_cards[table_img_num].card[card_num], 5);
	Mat element = getStructuringElement(MORPH_RECT, Size(3, 3), Point(-1, -1));
	erode(detected_cards[table_img_num].card[card_num],
detected_cards[table_img_num].card[card_num], element);
	detected_cards[table_img_num].thresholded_card[card_num] =
thresholding(detected_cards[table_img_num].card[card_num], 150, 255);
	bitwise_not(detected_cards[table_img_num].thresholded_card[card_num],
detected_cards[table_img_num].thresholded_card[card_num]);
}
```

wywołuje kilka funkcji filtrujących. Jednakże biorąc pod uwagę, że każdy z tych filtrów wykonywany jest za pomocą jednej linijki kodu, więc nieuzasadnionym byłoby tworzenie funkcji do wywołania tylko jednej innej funkcji. Dlatego zakładając, że pojedynczą odpowiedzialnością danej funkcji jest wywoływanie innych funkcji to kod spełnia to wymaganie. Inny przykład (funkcja obliczająca średnią wartość kanału R w konturze):

```
int mean_symbols_color(const Mat single_card, const vector<vector<Point>> contours,
int contour_num, vector<Vec4i> hierarchy)
{
	Mat mask = Mat::zeros(701, 451, CV_8U);
	drawContours(mask, contours, contour_num, Scalar(255, 255, 255), -1, 8,
hierarchy);
	Scalar card_symbol_color = mean(single_card, mask);
	return card_symbol_color.val[2];
```

```
}
```

- **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:

```
	vector<vector<Point>> contours;
	vector<Vec4i> hierarchy;
	int card_value = 0;
	string suit;
	float harmonic_mean = 0;
```

- **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:

```
vector<img_data> load_source_img_data()
{
	vector<String> pathes = images_paths();
	vector<img_data> table_images(pathes.size());
	for (int i = 0; i < pathes.size(); i++)
	{
		Mat current_image = imread(pathes[i]);
		table_images[i].table_img = current_image;
		string name = pathes[i];
		size_t char_number = name.find_last_of('/');
		name = name.substr(char_number + 1);
		table_images[i].img_name = name;
	}
	return table_images;
}
```

- **Komentarze i przypisy**

W kodzie oprócz nazwy funkcji informującej o jej działaniu oraz nazw zmiennych informujących o ich przeznaczeniu znajdują się również komentarze, które opisują większość funkcji. Ponadto komentarze pełnią funkcję porządkowania kodu (oddzielają od siebie poszczególne bloki). Przykład:

```
// opencv filters used to create correct thresholded image of card
void base_filters(vector<single_cards> &detected_cards, int table_img_num, int
card_num);

// calculates the average value of the red channel inside symbol contour
int mean_symbols_color(const Mat single_card, const vector<vector<Point>> contours,
int contour_num, vector<Vec4i> hierarchy);

// recognize symbol based on the ratio of the sides of the bounded rectangle and its
color
string symbols_recognition(const Mat single_card, const vector<vector<Point>>
contours, int contour_num, vector<Vec4i> hierarchy);
```

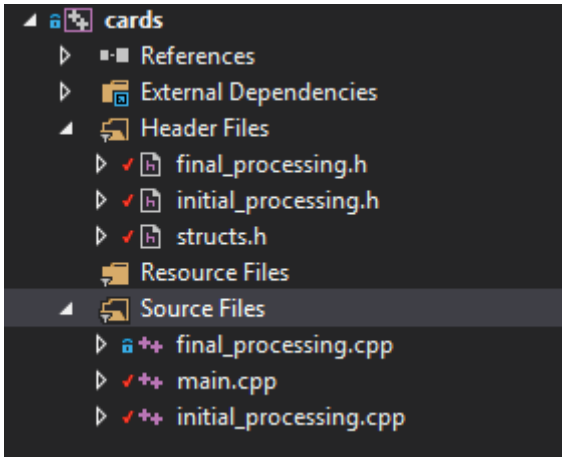
```
// load data in card structure
void saving_card_data(vector<single_cards> &detected_cards, const string card_suit,
const int card_value, int table_img_num, int card_number);

// calculate harmonic mean of the tanle image card
void harmonic_mean_calculation(vector<single_cards> &detected_cards);

// processing operations of single detected cards in the right order - find contours
bigger than x
void final_processing(vector<single_cards> &detected_cards);
```

- **Podział na interfejs i warstwę przetwarzania**

Cały projekt podzielony jest na 6 plików: main.cpp, structs.h, initial_processing.h, initial_processing.cpp, final_processing.h, final_processing.cpp. W pliku main.cpp jest jedynie wywołanie funkcji z pliku inital_processing oraz final_processing oraz jedna funkcja odpowiadająca za interfejs, który w tym programie ograniczony jest do wyświetlenia wszystkich informacji w konsoli po wykonaniu się programu. Dzięki takiemu podziałowi w żadnym z plików nie ma więcej niż 90 linijek co ułatwia odbiór. Natomiast nazwy plików jednoznacznie mówią czego możemy się w nich spodziewać. W pliku structs.h zadeklarowane są dwie struktury używane w programie. W plikach initial_processing znajdują się funkcje niezbędne do wykonywania wszystkich operacji do momentu segmentacji kart, natomiast w plikach final_processing zawarto funkcje odpowiedziane za pracę na pojedynczych kartach.



Rysunek 18

- **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 to umieszczony został w oddzielnej funkcji i wywoływany jest w pętli. Funkcje initial_processing() oraz final_processing() z pozoru mogą wyglądać podobnie, jednak ich wnętrze zawiera zasadnicze różnice uniemożliwiające ich połączenia (wywoływane funkcje oraz inicjalizowane zmienne są inne oraz znajdują się w różnych miejscach pętli), ponadto każda z tych funkcji jest innego typu: `void` oraz `vevctor<single_cards>`.

6. Opis teoretyczny

- **Opis teoretyczny problemu**

Krótki opis znajduje się w punkcie „Wstęp”.

Należało stworzyć program, który wykryje wartości kart i ich symbole na zdjęciach zawierających po 4 karty. Zdjęcia były poddane dodatkowej obróbce aby utrudnić zadanie i sprawić, że projekt będzie bardziej „realistyczny”, dodano rozmycie, szumy, gradient. Konieczne było dobranie takiego algorytmu, który radziłby sobie z problemem dla każdego przypadku. Jednakże podstawowym wyzwaniem było wymyślenie kryteriów, które program by stosował do odróżnienia kart od siebie (ich symboli).

W drugiej części należało obliczyć średnią harmoniczną – nie było to trudnością, mając rozwiązany poprzedni problem.

Średnią harmoniczną n liczb dodatnich $a_1, a_2, \ldots, a_n$ nazywamy liczbę:

$$\frac{n}{\frac{1}{a_1} + \frac{1}{a_2} + \cdots + \frac{1}{a_n}}$$

- **Projekt ścieżki przetwarzania**

Dokładny opis poszczególnych kroków znajduje się w części „Projekt rozwiązania”. Tutaj zaprezentowana zostanie po kolei, w skrócie, ścieżka przetwarzania w postaci punktów:

  - Wczytanie obrazów źródłowych
  - Progowanie/binaryzacja obrazów źródłowych
  - Wyszukanie konturów większych od zadanego progu (znalezienie konturów kart)
  - Przybliżenie konturów kart czworokątami
  - Posortowanie rogów czworokątów tak samo dla każdej karty
  - Korekta pespektywy – sprowadzenia kart do postaci prostokątów
  - Przepisanie danych do nowej struktury
  - Zastosowanie filtra medianowego dla obrazu każdej karty
  - Zastosowanie erozji dla obrazu każdej karty
  - Progowanie/binaryzacja obrazu każdej karty
  - Odwrócenie wartości bitów zbinaryzowanych obrazów każdej karty
  - Wyszukanie konturów większych od zadanego progu (znalezienie symboli)
  - Obliczenie średniej wartości kanału R dla wybranego symbolu z karty
  - Opisanie prostokąta na wybranym konturze symbolu z karty
  - Określenie symbolu na bazie barwy oraz stosunku boków prostokąta opisanego
  - Wpisanie informacji do struktury
  - Wypisanie obliczonych informacji w konsoli

- **Opis problemów i ich rozwiązania**

Jednym z problemów był fakt, że po progowaniu poszczególnych obrazów kart (zwłaszcza z rozmyciem lub szumem) w przypadku trefla i symbolu pik, ich „nóżki” nie były połączone z główną częścią symbolu. Skutecznym sposobem połączenia okazała się operacja erozji, która z małym elementem strukturalnym w niewielkim stopniu degradowała kształt symboli, a powiększenie ich o kilka pikseli umożliwiło ponowne złączenie obiektów w jeden symbol.

Kolejnym problemem było wyeliminowanie z konturów symboli pomocniczych w rogach kart, jednak ten problem udało się dość łatwo rozwiązać określając minimalną powierzchnię konturów – dzięki temu mniejsze kontury zostały wyeliminowane.

Z początku największe problemy były przewidywane przy obrazie z szumem, który znacząco wpływał na jakość zdjęcia, jednak dzięki zastosowaniu filtru medianowego, którego działanie idealnie nadaje się do usuwania pojedynczych pikseli szumu z obrazu udało się w dużej mierze wyeliminować wpływ szumu na wyniki.

- **Dobór algorytmów**

Do rozpoznawania symboli, jako jedna ze składowych algorytmu, była barwa symbolu, jest to dość oczywisty wybór, ponieważ jest ją łatwo sprawdzić i przy mniej więcej prawidłowych ustawieniach balansu bieli otrzymane wyniki zawsze są jednoznaczne i bardzo różniące się pomiędzy czerwonymi a czarnymi kartami. Drugim elementem algorytmu jest obliczenie stosunku boków opisanego na konturze symbolu prostokąta. Wybór akurat takiego sposobu podyktowany jest dużą łatwością w sprawdzeniu (gotowa funkcja w bibliotece OpenCV) oraz faktem, że symbole danego koloru mają wyraźnie odmienne wartości tego parametru. Można sobie wyobrazić jeszcze inne sposoby rozpoznania symboli jak np. stosunek pola powierzchni do obwodu konturu lub stosunek pola koła opisanego na konturze do pola konturu itp. Prawdopodobnie nie dałyby jednak one tak jednoznacznych rezultatów jak wybrana metoda.

- **Ocena działania wybranych algorytmów**

Wybrana metoda działa niezawodnie dla 20 zdjęć, dla których została stworzona, co ważne jest ona uniwersalna w ramach tych obrazów. Jednakże trzeba mieć na uwadze, że posługiwanie się tą metodą w stosunku do dowolnej bazy zdjęć (np. różne perspektywy, różne odległości od kamery) byłoby możliwe tylko po pewnych modyfikacjach. Przykładowo ustawione na „sztywno” pewne progi np. pola powierzchni konturów mogłyby nie zdać egzaminu przy zdjęciach z większej odległości lub w innej rozdzielczości. Jeszcze większą przeszkodą mogą być zdjęcia innej talii kart - drobne różnice w projektowaniu symboli przez różnych producentów dla człowieka nie mają znaczenia, ale przykładowo wystarczyłoby, że stosunek boków danych symboli nie byłby zachowany i algorytm nie działałby prawidłowo.

Podsumowując program spełnia wszystkie postawione mu w tym projekcie wymagania, jednak nie można by go prawdopodobnie stosować jako system wizyjny komputerowego gracza.

7. Bibliografia

- https://pl.wikipedia.org/wiki/%C5%9Arednia_harmoniczna
- https://docs.opencv.org