

Zestawu filtrów i efektów do przetwarzania obrazów z kamery internetowej - sprawozdanie.

Piotr Błądek

12 kwietnia 2015

Spis treści

1	Technologia	2
2	Zewnętrzne biblioteki	3
3	Działanie aplikacji	4
4	Problemy podczas pisania aplikacji	4
5	Uruchomienie aplikacji	5
6	Filtry i efekty	6
6.1	Obraz w skali szarości	6
6.2	Inwersja kolorów	6
6.3	Odbicie symetryczne obrazu względem osi	7
6.4	Zmiana kontrastu	8
6.5	Zmiana jasności	8
6.6	Gamma	9
6.7	Solaryzacja	10
6.8	Wyoszczepienie krawędzi	11
6.9	Dodanie ramki	11
6.10	Segmentacja obrazu	12
6.11	Rozmycie Gaussa	13
6.12	Erozja, czyli zwężanie	13
6.13	Rozszerzanie, czyli dylatacja	14
7	Kilka zrzutów ekranu z aplikacji	14

Listings

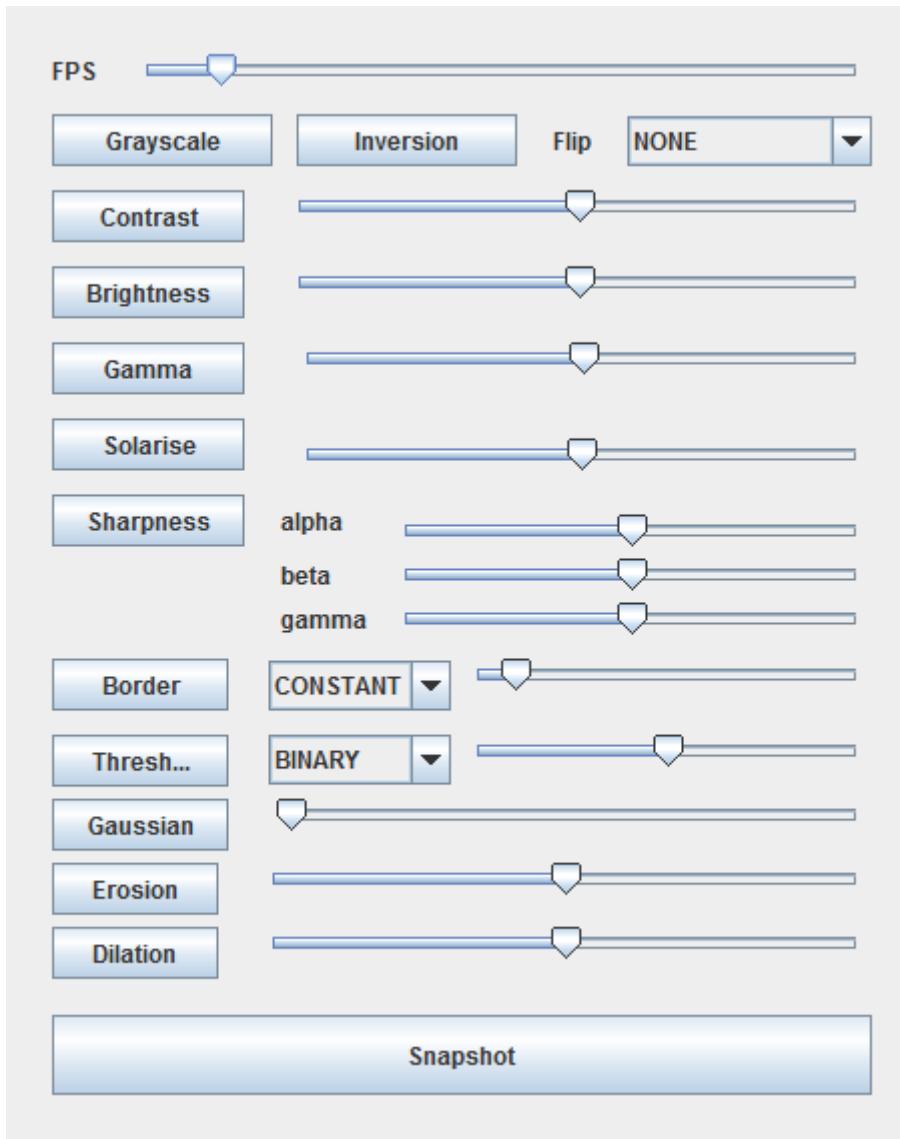
1	Funkcja zmiany obrazu na obraz w skali szarości.	6
2	Funkcja inwertująca kolory obrazu.	7
3	Funkcja odbijająca obraz względem osi.	7
4	Funkcja zmieniająca kontrast obrazu.	8
5	Funkcja zmieniająca jasność obrazu.	9
6	Funkcja zmieniająca gammę obrazu.	9
7	Funkcja zmieniająca solaryzację obrazu.	10
8	Funkcja zmieniająca wyrazistość krawędzi.	11
9	Funkcja dodająca ramkę do obrazu.	11
10	Funkcja segmentująca obraz.	12
11	Funkcja rozmycia obrazu.	13
12	Funkcja dodająca efekt erozji do obrazu.	13
13	Funkcja dodająca efekt rozszerzenia do obrazu.	14

Spis rysunków

1	Przyciski funkcyjne aplikacji.	3
2	Okno zapisu migawki na dysk.	4
3	Okno oglądu obrazu z kamery.	5
4	Obraz bez włączonych efektów, prosto z kamery.	15
5	Segmentacja binarna obrazu, zmiana jasności oraz obrócenie względem osi X.	15
6	Inwersja barw obrazu.	16
7	Dodanie ramki typu REPLICATE i wyostrzenie krawędzi obrazu.	16
8	Solaryzacja obrazu.	17
9	Erozja oraz solaryzacja z mniejszym progiem niż wyżej (widać różnicę).	17
10	Obraz z włączonym efektem rozszerzania i skali szarości.	18
11	Zapisywanie migawki na dysk.	18

1 Technologia

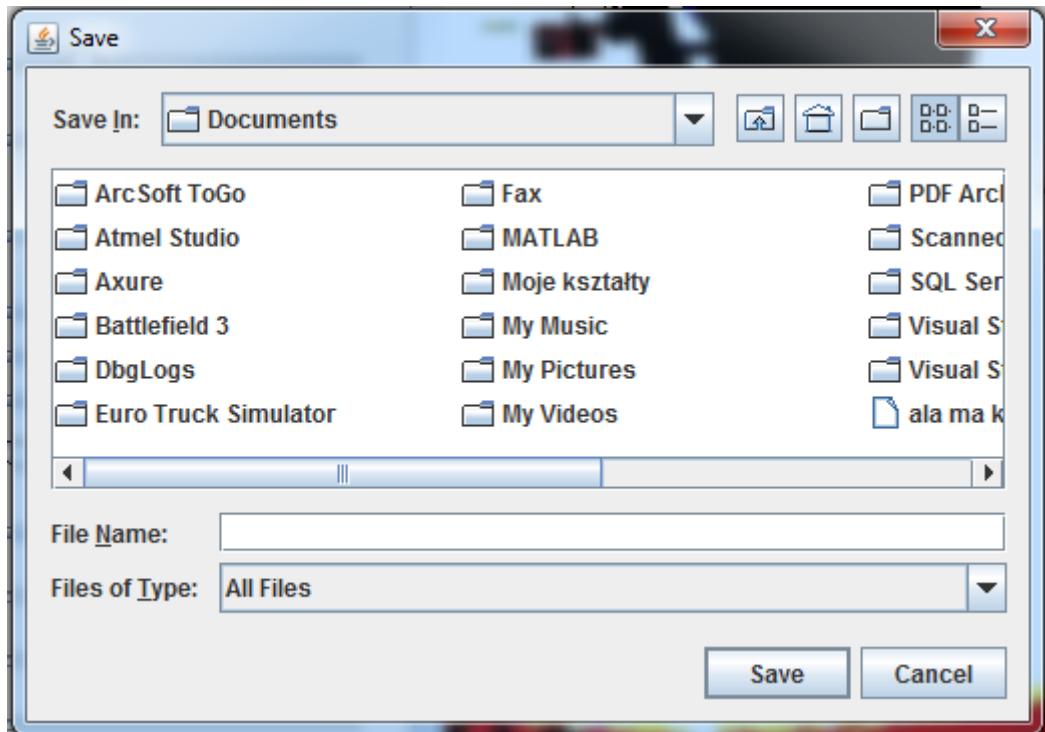
W ramach projektu została napisana aplikacja typu desktop w języku Java z GUI w frameworku Swing.



Rysunek 1: Przyciski funkcjonalne aplikacji.

2 Zewnętrzne biblioteki

Przy pisaniu projektu skorzystałem z dwóch zewnętrznych bibliotek, jedna (LTI-CIVIL) służy do obsługi kamer internetowych, natomiast drugą jest biblioteka efektów graficznych OpenCV, wykorzystałem ją do przetworzenia obrazu z kamery tak aby uzyskać konkretne filtry i efekty które samemu byłboby trudno uzyskać a ich implementacja byłaby czasochłonna. Sam zaimplementowałem mnóstwo więcej połowej funkcji, przy reszcie skorzystałem z pomocy wspomnianej wcześniej biblioteki.



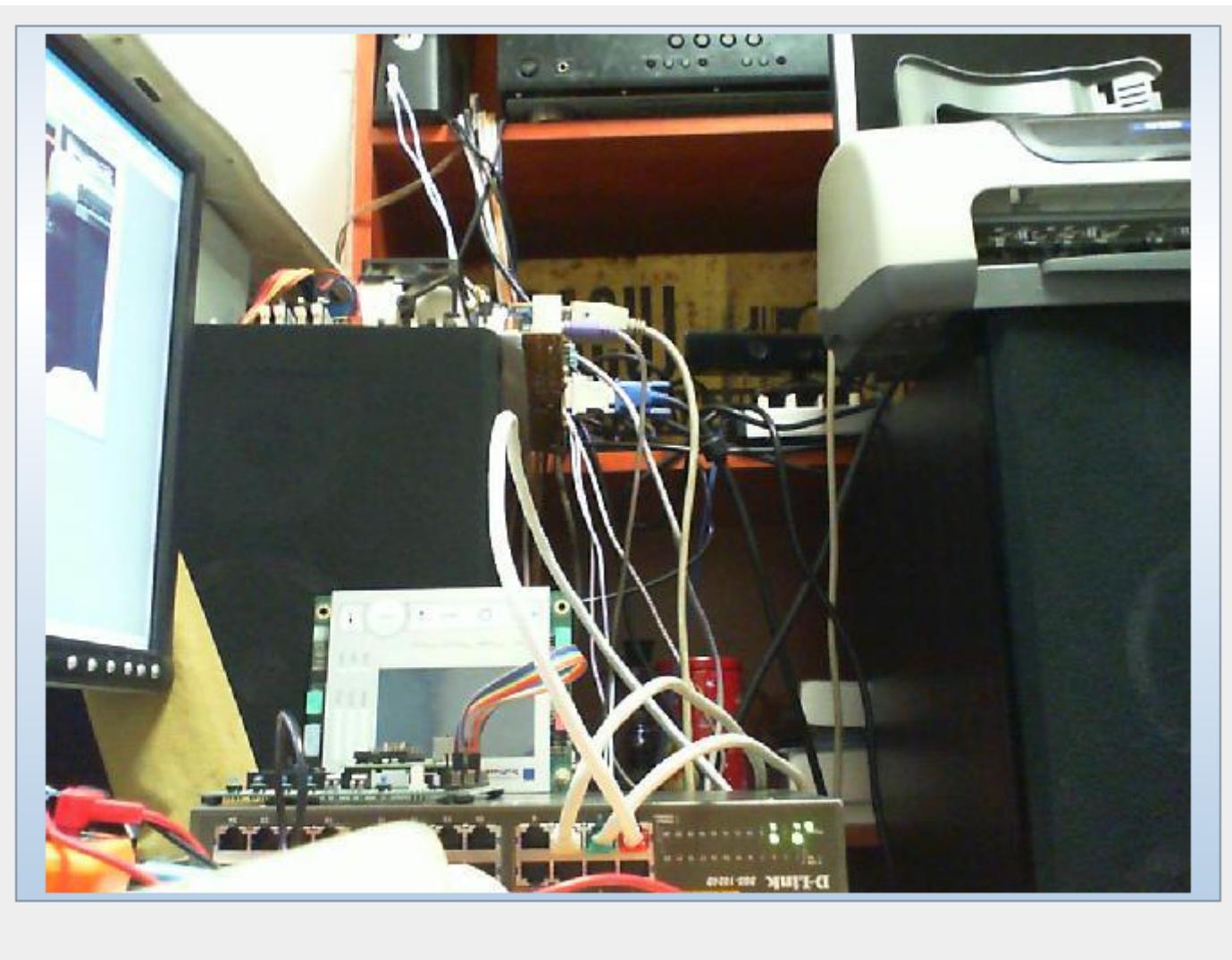
Rysunek 2: Okno zapisu migawki na dysk.

3 Działanie aplikacji

Aplikacja korzysta z kamery internetowej w celu pobrania z niej obrazu i wyświetlenia go na ekranie. Aplikacja udostępnia zestaw kontrolek (Rysunek 1.) które zmieniają obraz w zależności od wybranych opcji, obraz jest poddawany przekształceniom "w locie" to znaczy że to co użytkownik wybiera jest natychmiast widoczne na obrazie z kamery. Aplikacja umożliwia zapisanie obrazu w postaci migawki na dysku (Rysunek 2.).

4 Problemy podczas pisania aplikacji

Podczas pisania aplikacji napotkałem kilka problemów z którymi na szczęście udało mi się szybko uporać. Najbardziej uporczywym problemem okazały się biblioteki. Obie nie były w pierwotnej wersji pisane do współpracy z Java, natomiast zostały później do tego przystosowane. Mankamentem biblioteki LTI-CIVIL jest to że pracuje jedynie z Java x86 czyli w 32 bitowej wersji, a więc jeżeli chcemy uruchomić program musimy to zrobić z użyciem 32 bitowej maszyny wirtualnej javy. Druga wspomniana biblioteka to OpenCV, problemem tej biblioteki jest to że została napisana dla języka C++ i przystosowana do korzystania z niej w Javie. Podstawowym problemem jest to że nie możemy jej podawać Javowych klas przechowujących obraz, trzeba jej podać ścieżkę do obrazu który ona sobie przeczyta i zamieni w miejscu. To znaczy że przy korzystaniu z jej usług



Rysunek 3: Okno oglądu obrazu z kamery.

musiałem odczytać obraz z kamery, następnie zapisać go na dysk, wywołać funkcję biblioteki i podać jej ścieżkę do obrazu (biblioteka odczytywała obraz, przetwarzała go i zapisywała do tego samego miejsca), a następnie odczytać obraz z dysku, jak widać nie jest to rozwiązańe optymalne i przy korzystaniu w funkcji które zostały zaimplementowane przy użyciu biblioteki OpenCV widać że są one bardziej mozolne i potrzebują więcej czasu.

5 Uruchomienie aplikacji

Aby aplikację uruchomić musimy posiadać wersję 32x Javy, wywołanie odbywa się poprzez polecenie:

```
java -jar Kamerka.jar
```

Powinno pojawić się okno aplikacji, aplikacja sama wykryje kamerę w systemie operacyjnym (jeżeli jest dwie lub więcej kamer w systemie aplikacja wybierze jedną z nich, tą która jest na mniejszym numerze portu USB) i zacznie przechwytywać z niej obraz.

6 Filtry i efekty

W aplikacji zostały zaimplementowane następujące filtry i efekty:

6.1 Obraz w skali szarości

Aby zamienić obraz do skali szarości zastosowałem poniższą funkcję. Funkcja ta przyjmuje jako argument obiekt klasy BufferedImage, podobnie jak i inne funkcje które implementowałem sam. Jedną z metod zamiany na skalę szarości jest dodanie wartości wszystkich kolorów, podzielenie jej przez ich ilość czyli trzy i przypisanie wyniku do wartości wszystkich kolorów.

```
public void grayscale(BufferedImage myImage) {  
    for (int i = 0; i < myImage.getHeight(); i++) {  
        for (int j = 0; j < myImage.getWidth(); j++) {  
            Color c = new Color(myImage.getRGB(j, i));  
            int red = (int) (c.getRed());  
            int green = (int) (c.getGreen());  
            int blue = (int) (c.getBlue());  
            int avg = (red + green + blue) / 3;  
            Color newColor = new Color(avg, avg, avg);  
            myImage.setRGB(j, i, newColor.getRGB());  
        }  
    }  
}
```

Listing 1: Funkcja zmiany obrazu na obraz w skali szarości.

6.2 Inwersja kolorów

Inwersja kolorów jest równie prostym procesem jak wcześniejsza skala szarości. Żeby zrobić inwersję barw musimy z każdego piksela pobrać kolory składowych RGB i od liczby 255 (maksymalna wartość koloru) odjąć wartość aktualną, w ten sposób dostaniemy obraz o odwróconych kolorach.

```

public void inversion(BufferedImage myImage) {
    for (int i = 0; i < myImage.getHeight(); i++) {
        for (int j = 0; j < myImage.getWidth(); j++) {
            Color c = new Color(myImage.getRGB(j, i));
            int red = (int) (c.getRed());
            int green = (int) (c.getGreen());
            int blue = (int) (c.getBlue());
            int newRed = 255 - red;
            int newGreen = 255 - green;
            int newBlue = 255 - blue;
            Color newColor = new Color(newRed, newGreen, newBlue);
            myImage.setRGB(j, i, newColor.getRGB());
        }
    }
}

```

Listing 2: Funkcja inwertująca kolory obrazu.

6.3 Odbicie symetryczne obrazu względem osi

Przy odbiciu skorzystałem już z usług biblioteki openCV. Samo obrócenie obrazu można było łatwo napisać samemu, polega ono na tym że zamieniamy pierwszy wiersz obrazu z ostatnim, drugi z przedostatnim i tak dalej, aż dojdziemy do środka. Parametrem code definiujemy sposób odbicia (albo względem osi X, albo osi Y).

```

public void flip(String sourcePath, int code){
    if (code == -1) return;
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
    Mat destination = new Mat(source.rows(), source.cols(), source.type());
    destination = source;
    Core.flip(source, source, code);
    Highgui.imwrite(sourcePath, destination);
}

```

Listing 3: Funkcja odbijająca obraz względem osi.

6.4 Zmiana kontrastu

Pierwszym krokiem zmiany kontrastu obrazu jest wyliczenie współczynnika kontrastu, który liczę według poniższego wzoru, parametr contrast jest wybraną wartością kontrastu:

$$factor = \frac{259 * (contrast + 255)}{255 * (259 - contrast)}$$

Nowy kolor tworzymy zamieniając składową RGB każdego piksela wynikiem równania:

$$newColor = factor * (color - 128) + 128$$

Dodatkowo funkcja truncate dba o to żeby wyliczona wartość mieściła się w zakresie 0-255.

```
public void contrast(BufferedImage myImage, int contrast) {  
    int f = (259 * (contrast + 255));  
    int g = (255 * (259 - contrast));  
    double factor = (double) f / g;  
    for (int i = 0; i < myImage.getHeight(); i++) {  
        for (int j = 0; j < myImage.getWidth(); j++) {  
            Color c = new Color(myImage.getRGB(j, i));  
            int red = (int) (c.getRed());  
            int green = (int) (c.getGreen());  
            int blue = (int) (c.getBlue());  
            int newRed = truncate(factor * (red - 128) + 128);  
            int newGreen = truncate(factor * (green - 128) + 128);  
            int newBlue = truncate(factor * (blue - 128) + 128);  
            Color newColor = new Color(newRed, newGreen, newBlue);  
            myImage.setRGB(j, i, newColor.getRGB());  
        }  
    }  
}
```

Listing 4: Funkcja zmieniająca kontrast obrazu.

6.5 Zmiana jasności

Funkcja zmiany jasności polega po prostu na dodaniu do aktualnej wartości składowej koloru, wybranej na suwaku wartości współczynnika brightness. Obraz rozjaśnia się (albo ściemnia przy ujemnych wartościach parametru brightness) dlatego że przy dążeniu wartości koloru do 255 obraz staje się coraz bardziej bielszy (wartość koloru 255 to kolor biały), symetrycznie przebiega ściemnianie (0 to kolor czarny). Podobnie jak wcześniejsza funkcja truncate zapewnia nam że wartość koloru będzie się mieściła w zakresie 0-255.

```

public void brightness(BufferedImage myImage, int brightness) {
    for (int i = 0; i < myImage.getHeight(); i++) {
        for (int j = 0; j < myImage.getWidth(); j++) {
            Color c = new Color(myImage.getRGB(j, i));
            int red = (int) (c.getRed());
            int green = (int) (c.getGreen());
            int blue = (int) (c.getBlue());
            int newRed = truncate(red + brightness);
            int newGreen = truncate(green + brightness);
            int newBlue = truncate(blue + brightness);
            Color newColor = new Color(newRed, newGreen, newBlue);
            myImage.setRGB(j, i, newColor.getRGB());
        }
    }
}

```

Listing 5: Funkcja zmieniająca jasność obrazu.

6.6 Gamma

Wartości składowych RGB pikseli dla korekcji gamma liczy się ze wzoru:

$$newColor = 255 * \left(\frac{oldColor}{255} \right)^{\frac{1}{\gamma}}$$

Gdzie współczynnik gamma w przypadku mojej aplikacji wynosi:

$$0.01 \leq \gamma \leq 7.99$$

```

public void gamma(BufferedImage myImage, int g) {
    double gamma = (double) g / 100;
    double gC = (double) 1 / gamma;
    for (int i = 0; i < myImage.getHeight(); i++) {
        for (int j = 0; j < myImage.getWidth(); j++) {
            Color c = new Color(myImage.getRGB(j, i));
            int red = (int) (c.getRed());
            int green = (int) (c.getGreen());
            int blue = (int) (c.getBlue());
            int newRed = (int) (255 * Math.pow(((double) red / 255), gC));

```

```

        int newGreen = (int) (255 * Math.pow(((double) green / 255), gC));
        int newBlue = (int) (255 * Math.pow(((double) blue / 255), gC));
        Color newColor = new Color(newRed, newGreen, newBlue);
        myImage.setRGB(j, i, newColor.getRGB());
    }
}
}

```

Listing 6: Funkcja zmieniająca gammę obrazu.

6.7 Solaryzacja

Solaryzacja jest efektem zbliżonym do inwersji kolorów, różnica jest taka że korzystamy tu z parametru threshold który rozgranicza nam wartość koloru który ma zostać zinwertowany, w moim przypadku wszystkie kolory które są poniżej parametru threshold zostają zamienione na przeciwnie, kolory o większej wartości pozostają niezmienione. Wartość threshold ustala się za pomocą suwaka dostępnego obok przycisku włączającego solaryzację.

```

public void solarise(BufferedImage myImage, int threshold) {
    for (int i = 0; i < myImage.getHeight(); i++) {
        for (int j = 0; j < myImage.getWidth(); j++) {
            Color c = new Color(myImage.getRGB(j, i));
            int red = (int) (c.getRed());
            int green = (int) (c.getGreen());
            int blue = (int) (c.getBlue());
            int newRed = red;
            int newGreen = green;
            int newBlue = blue;
            if (red < threshold) newRed = 255 - red;
            if (green < threshold) newGreen = 255 - green;
            if (blue < threshold) newBlue = 255 - blue;
            Color newColor = new Color(newRed, newGreen, newBlue);
            myImage.setRGB(j, i, newColor.getRGB());
        }
    }
}

```

Listing 7: Funkcja zmieniająca solaryzację obrazu.

6.8 Wyostrzenie krawędzi

W tej funkcji korzystamy z funkcji biblioteki OpenCV, która zwiększy nam ostrość krawędzi obrazu. Zastosowanie tego filtru oznacza, że do obliczenia nowej wartości punktu brane są pod uwagę wartości punktów z jego otoczenia. Każdy piksel z otoczeniem wnosi swój wkład - wagę podczas przeprowadzania obliczeń. Wagi te zapisywane są w postaci maski. Typowe rozmiary masek to 3 x 3, 5 x 5 bądź 7 x 7. Rozmiary masek są z reguły nieparzyste ponieważ piksel na środku reprezentuje piksel dla którego wykonywana jest operacja przekształcania filtrem. Filtry dolnoprzepustowe przepuszczają elementy obrazu o małej częstotliwości. Elementy o wysokiej częstotliwości (szumy, drobne szczegóły) są natomiast tłumione bądź wręcz blokowane. Wynikiem działania takich filtrów jest zredukowanie szumów, w szczególności gdy jest on jedno, dwupikslowy ale również wygładzenie i rozmycie obrazu. Filtry górnoprzepustowe przepuszczają i wzmacniają elementy obrazu o dużej częstotliwości, są to szumy, drobne szczegóły i krawędzie. Tłumieniu natomiast ulegają elementy o niskiej częstotliwości. Wynikiem działania takich filtrów jest wyostrzenie obrazu, a także zwiększenie ilości szumów. Parametr alpha to waga najważniejszych elementów maski, beta to waga drugorzędnych elementów maski (pobocznych).

```
public void sharpness(String sourcePath, int a, int b, int g) {
    double alpha = (double) a / 100;
    double beta = (double) b / 100;
    double gamma = (double) g / 100;
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
    Mat destination = new Mat(source.rows(), source.cols(), source.type());
    Imgproc.GaussianBlur(source, destination, new Size(0, 0), 10);
    Core.addWeighted(source, alpha, destination, beta, gamma, destination);
    Highgui.imwrite(sourcePath, destination);
}
```

Listing 8: Funkcja zmieniająca wyrazistość krawędzi.

6.9 Dodanie ramki

Biblioteka LTI-CIVIL udostępnia zestaw ramek które możemy dodać do swojego obrazu, parametr borderType to stała zdefiniowana w bibliotece która mówi nam jaki typ ramki dostaniemy. Zdefiniowanych jest 6 typów ramek, za pomocą suwaka możemy zwiększać szerokość ramki. Sama ramka nie nachodzi na obraz lecz jest dodawana do niego na zewnątrz, przez co obraz zostaje powiększony o dwie szerokości ramki w pionie i poziomie.

```
public void border(String sourcePath, int s, int borderType) {
    if (borderType == -1) {
```

```

        return ;
    }
    if ( borderType == 5) {
        borderType = 16;
    }
    double size = (double) s / 100;
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
    Mat destination = new Mat(source.rows(), source.cols(), source.type());
    int top, bottom, left, right;
    top = (int) (size * source.rows());
    bottom = (int) (size * source.rows());
    left = (int) (size * source.cols());
    right = (int) (size * source.cols());
    destination = source;
    Imgproc.copyMakeBorder(
        source, destination, top, bottom, left, right, borderType);
    Highgui.imwrite(sourcePath, destination);
}

```

Listing 9: Funkcja dodająca ramkę do obrazu.

6.10 Segmentacja obrazu

Segmentacja to podział obrazu na obszary odpowiadające poszczególnym, widocznym na obrazie obiektem. Twarzyszy temu indeksacja (etykietowanie) obiektów, czyli przypisanie każdemu obiekowi innej etykiety (wszystkie piksele danego obiektu otrzymują tę samą wartość – etykietę). Segmentacja podobnie jak wcześniej wyostrzanie krawędzi korzysta z masek do podziału obrazu na obszary, w efekcie wszystkie obiekty należące do tego samego obszaru otrzymują taką samą barwę. Biblioteka udostępnia nam pięć rodzajów segmentacji które zmienia się przy pomocy parametru type dostępnego z rozwijanej listy, progi segmentacji określa się za pomocą zmiennej thresh widocznej w GUI pod postacią suwaka.

```

public void thresholding(String sourcePath, int thresh, int type) {
    if (type == -1) {
        return;
    }
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);

```

```

    Mat destination = new Mat(source.rows(), source.cols(), source.type());
    destination = source;
    Imgproc.threshold(source, destination, thresh, 255, type);
    Highgui.imwrite(sourcePath, destination);
}

```

Listing 10: Funkcja segmentująca obraz.

6.11 Rozmycie Gaussa

Rozmycie to proces odwrotny do wyostrzania krawędzi, stosuje się tu filtr dolnoprzepustowy opisany szerzej powyżej, parametr size określa wielkość rozmęcia.

```

public void gaussian(String sourcePath, int size){
    try {
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
        Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
        Mat destination = new Mat(source.rows(), source.cols(), source.type());
        destination = source;
        if (size%2==0) size++;
        Imgproc.GaussianBlur(source, destination, new Size(size, size), 0);
        Highgui.imwrite(sourcePath, destination);
    } catch (Exception e) {
        System.out.println("Error in gaussian func" + e.getMessage());
    }
}

```

Listing 11: Funkcja rozmęcia obrazu.

6.12 Erozja, czyli zwężanie

Erozja, czyli zwężanie, jest zastosowaniem różnicy Minkowskiego do obrazu. Jest jednym z podstawowych przekształceń morfologicznych. Jej działanie polega na obcinaniu brzegów obiektu na obrazie. parametr size określa wielkość erozji.

```

public void erosion(String sourcePath, int size){
    System.loadLibrary( Core.NATIVE_LIBRARY_NAME );
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
    Mat destination = new Mat(source.rows(), source.cols(), source.type());
}

```

```

destination = source;
Mat element = Imgproc.getStructuringElement(Imgproc.MORPH_RECT,
    new Size(2*size + 1, 2*size+1));
Imgproc.erode(source, destination, element);
Highgui.imwrite(sourcePath, destination);
}

```

Listing 12: Funkcja dodająca efekt erozji do obrazu.

6.13 Rozszerzanie, czyli dylatacja

Dylatacja służy do zamykania małych otworów oraz zatok we wnętrzu figury. Obiekty zwiększają swoją objętość i jeśli dwa lub więcej obiektów położonych jest blisko siebie, zrastają się w większe obiekty. Rozszerzanie jest procesem bliźniaczym do erozji. Parametr size zwiększa wielkość dylatacji.

```

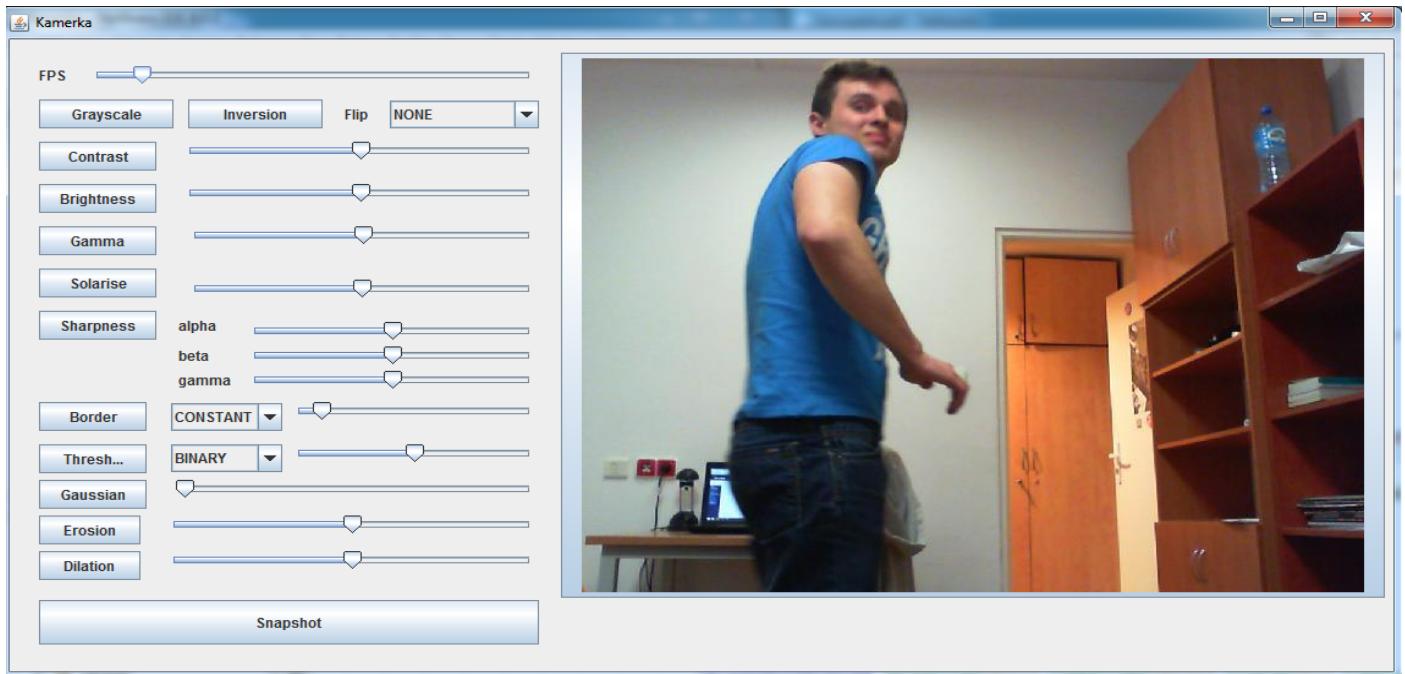
public void dilation(String sourcePath, int size){
    System.loadLibrary(Core.NATIVE_LIBRARY_NAME);
    Mat source = Highgui.imread(sourcePath, Highgui.CV_LOAD_IMAGE_COLOR);
    Mat destination = new Mat(source.rows(), source.cols(), source.type());
    destination = source;
    Mat element = Imgproc.getStructuringElement(Imgproc.MORPH_RECT,
        new Size(2*size + 1, 2*size+1));
    Imgproc.dilate(source, destination, element);
    Highgui.imwrite(sourcePath, destination);
}

```

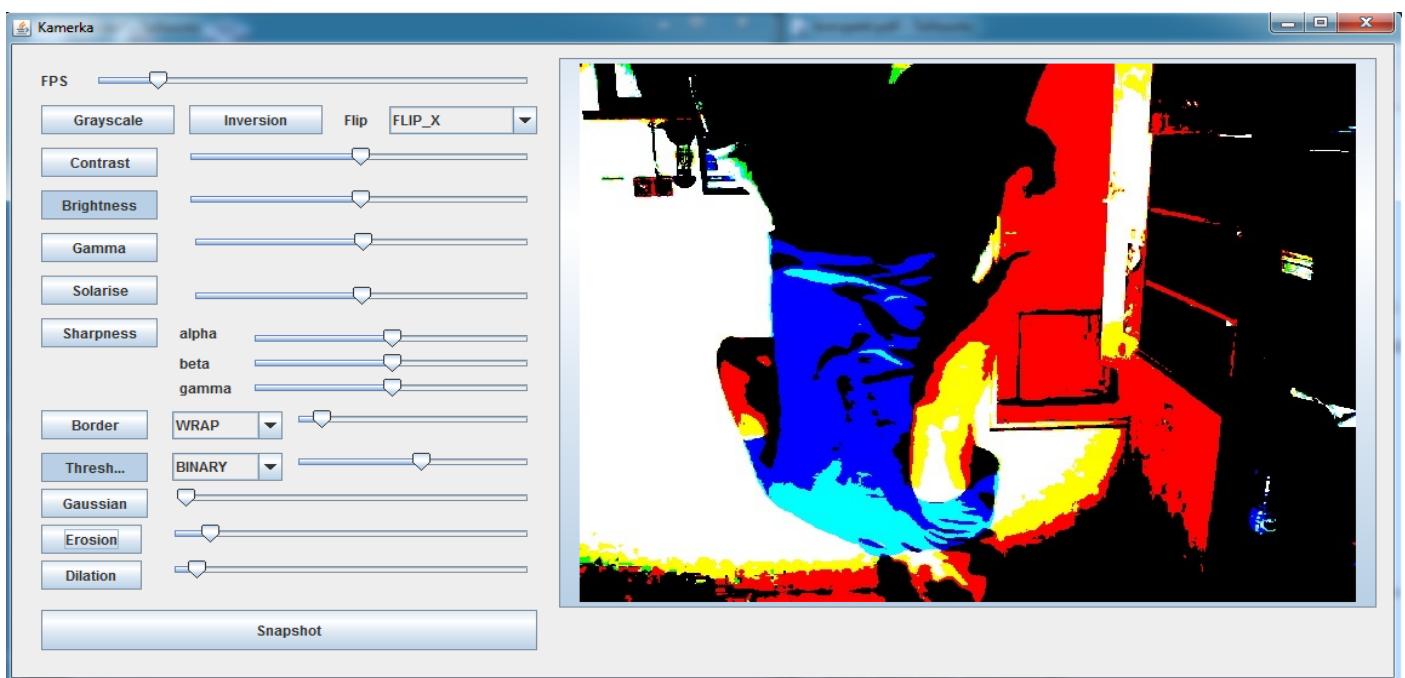
Listing 13: Funkcja dodająca efekt rozszerzenia do obrazu.

7 Kilka zrzutów ekranu z aplikacji

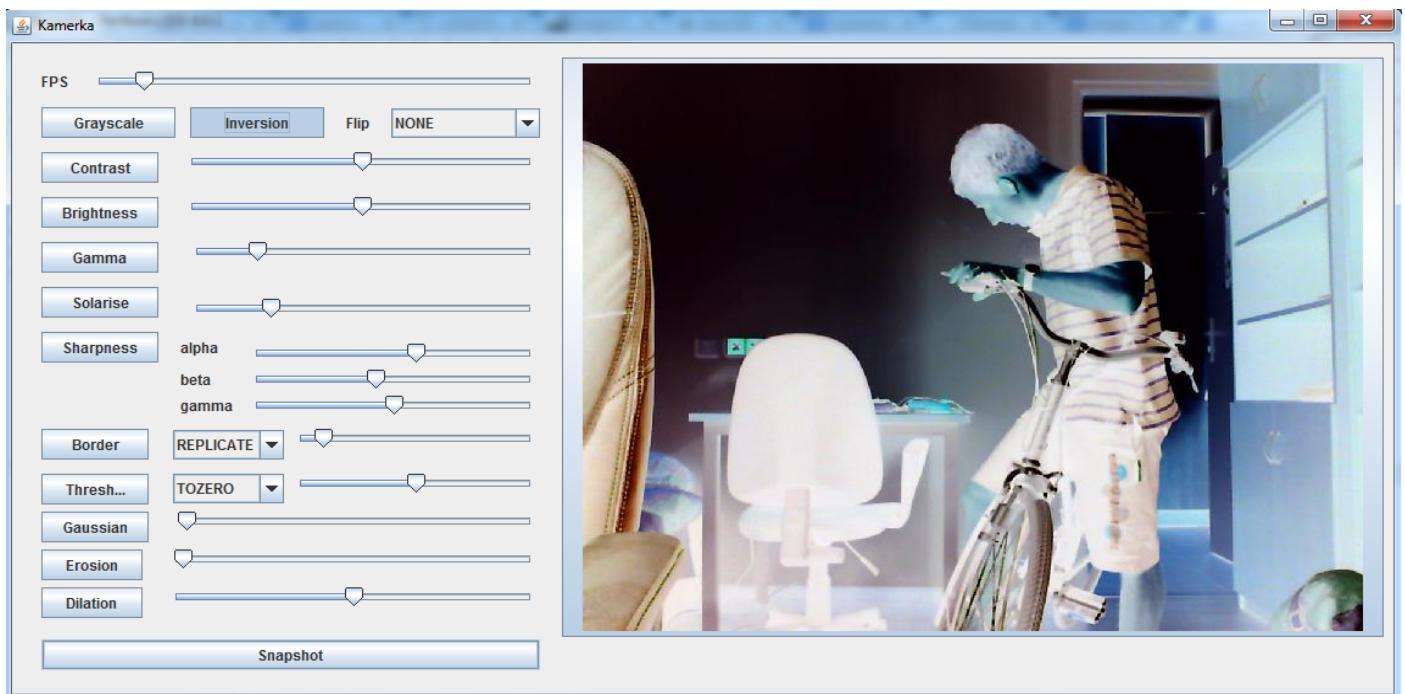
Poniżej zamieszczam kilka zrzutów ekranu z działania aplikacji (kolega widoczny na zrzutach nie zgodził się na ich rozpowszechnianie, ale wyraził zgodę na umieszczenie ich w sprawozdaniu).



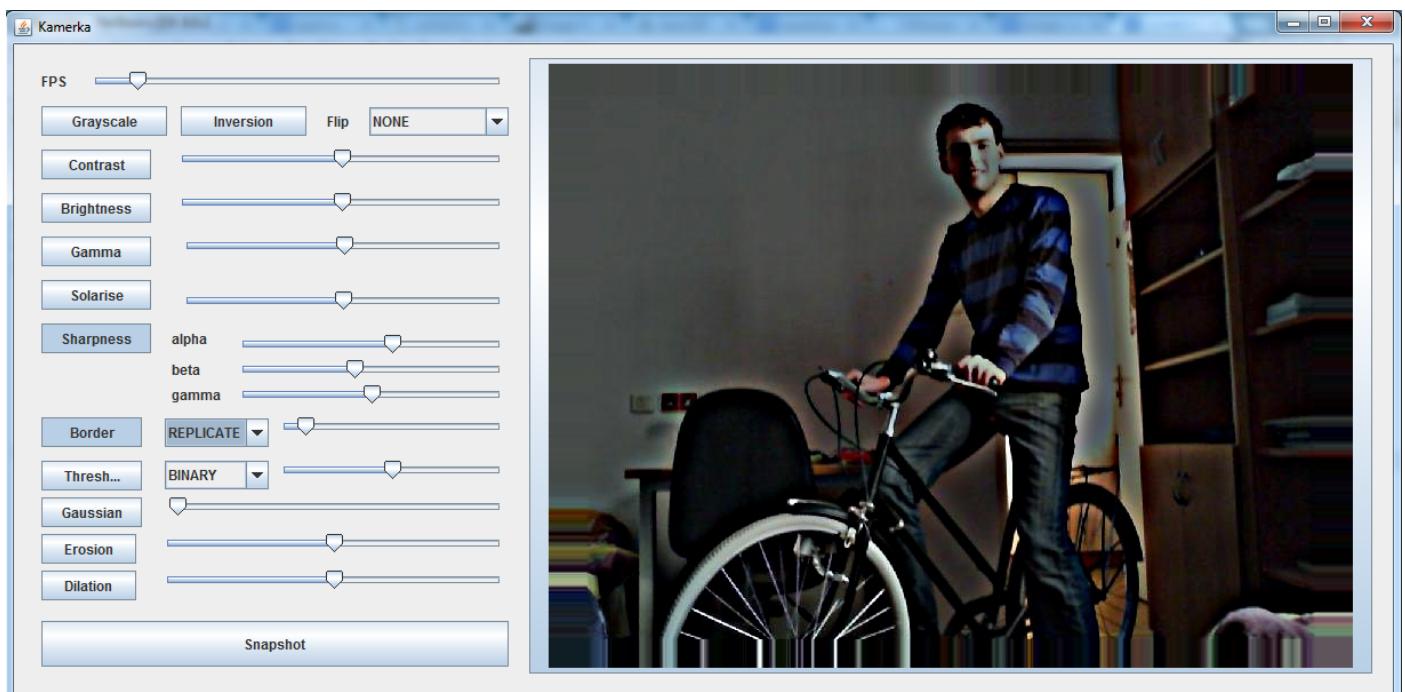
Rysunek 4: Obraz bez włączonych efektów, prosto z kamery.



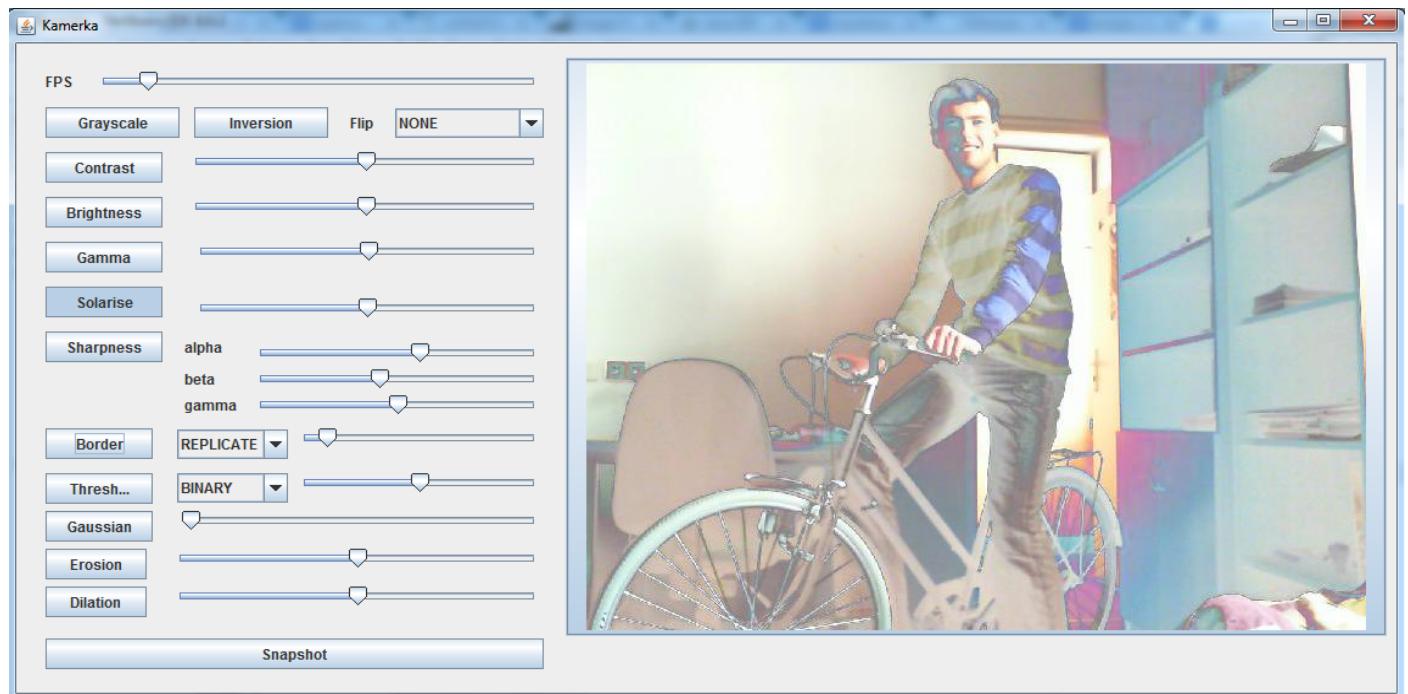
Rysunek 5: Segmentacja binarna obrazu, zmiana jasności oraz obrócenie względem osi X.



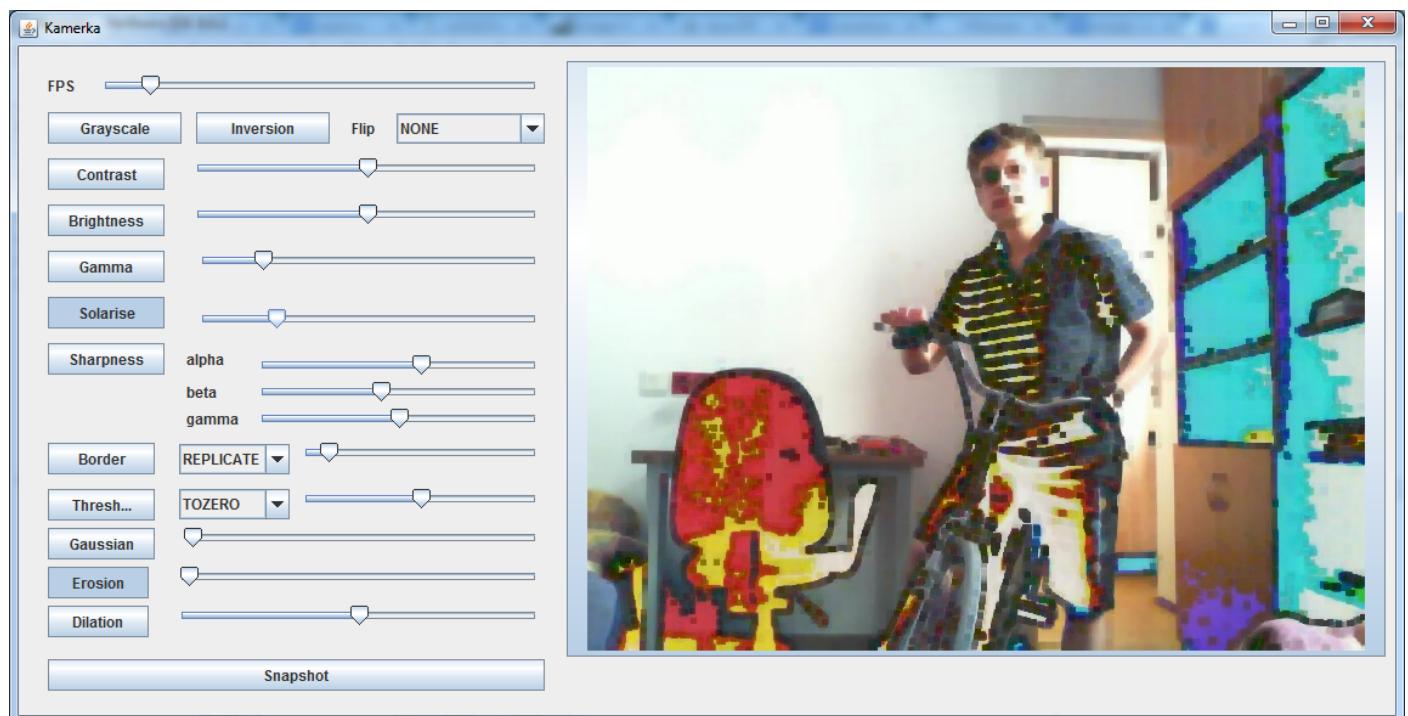
Rysunek 6: Inwersja barw obrazu.



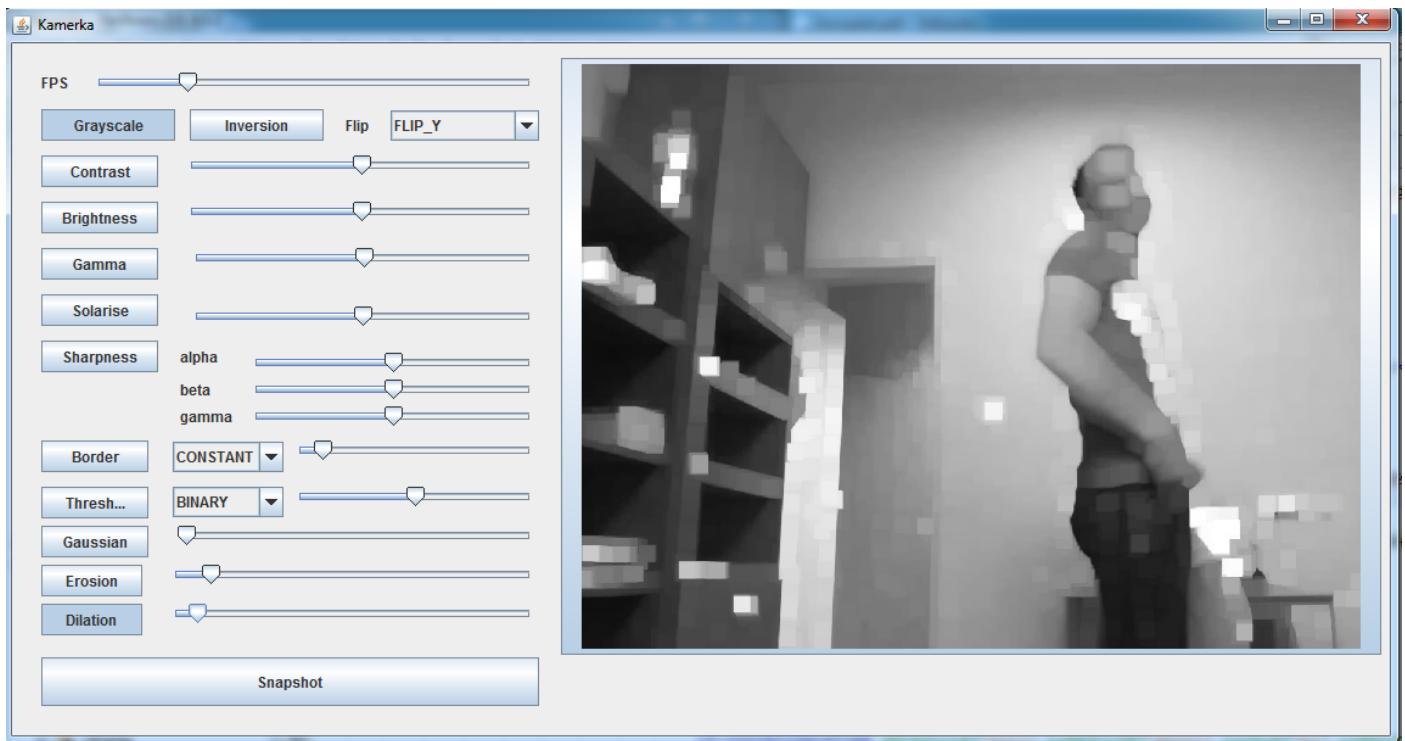
Rysunek 7: Dodanie ramki typu REPLICATE i wyostrzenie krawędzi obrazu.



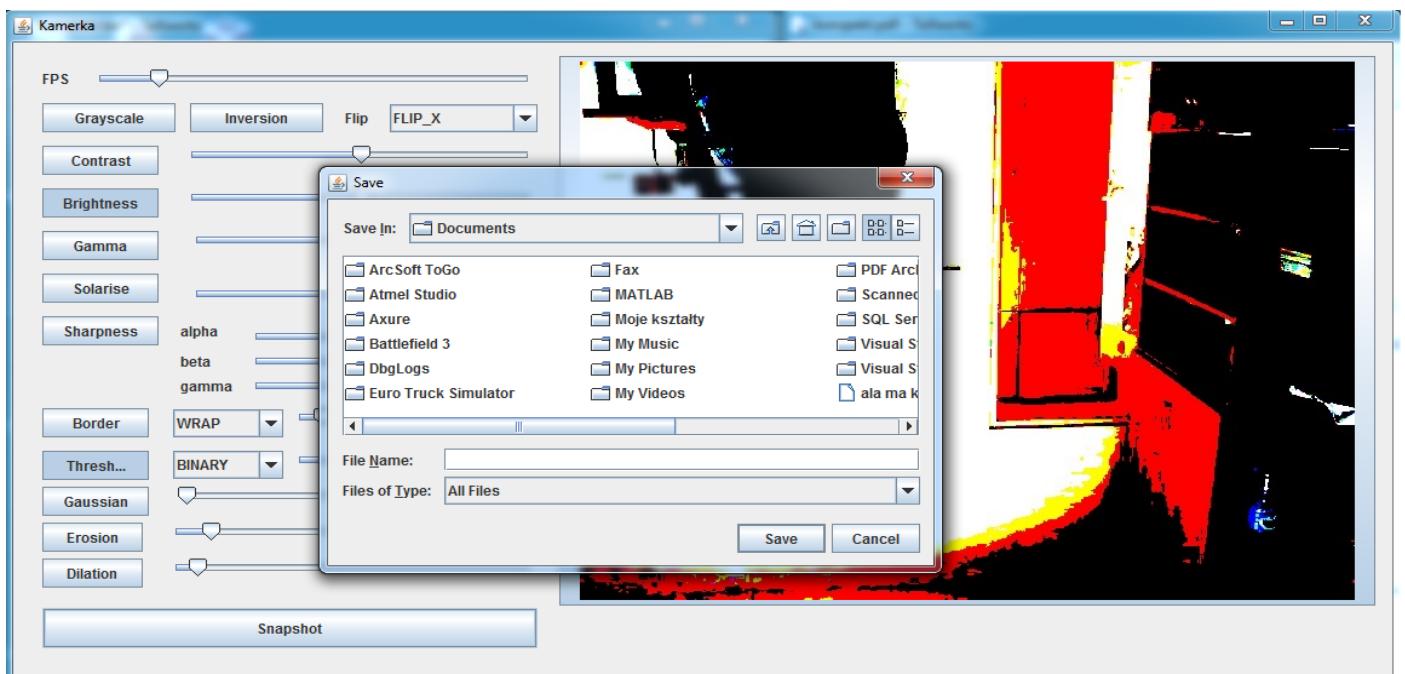
Rysunek 8: Solaryzacja obrazu.



Rysunek 9: Erozja oraz solaryzacja z mniejszym progiem niż wyżej (widać różnicę).



Rysunek 10: Obraz z włączonym efektem rozszerzania i skali szarości.



Rysunek 11: Zapisywanie migawki na dysk.