

Projekt indywidualny

Wykrywanie krawędzi na obrazach

Nikodem Wójcik

1 Opis projektu

Tematem mojego projektu indywidualnego było napisanie programu realizującego detekcję krawędzi na obrazach. Rozwiązanie realizujące założenia projektu oparłem na algorytmie Canny. Implementacja mojego rozwiązania została napisana w języku dotnet.

2 Wstęp teoretyczny

Algorytm Canny jest to wielostopniowy algorytm detekcji krawędzi na obrazie. Algorytm ten został opracowany w 1986 roku a jego autorem jest John F. Canny. Głównymi założeniami algorytmu była dobra detekcja krawędzi, ich umiejscowienie oraz brak tworzenia fałszywych krawędzi na nowo powstałym obrazie. Algorytm Canny zasadniczo składa się z czterech kroków:

1. Redukcja szumu
2. Szukanie natężenia gradientu obrazu
3. Usuwanie niemaksymalnych pikseli
4. Progowanie z histerezą

W dalszej części raportu opiszę poszczególne kroki algorytmu, przedstawię zarys implementacji danego kroki oraz obraz jaki został utworzony w danej fazie algorytmu.

Poniżej przedstawiam obraz jaki będę używał w prezentacji działania algorytmu. Jest on w rozdzielcości 1920x1280 oraz został pobrany z otwartych źródeł.



3 Realizacja poszczególnych kroków algorytmu

3.1 Redukcja szumu

Pierwszym krokiem algorytmu jest dokonanie redukcji szumu na zadanym obrazie. Wykrywanie krawędzi za pomocą algorytmu Canny jest wrażliwe na obecność szumów co może skutkować przetworzeniem mało istotnych krawędzi i doprowadzić do nieczytelności wytworzzonego obrazu. Przykładem zaobserwowania takiego zjawiska może być nadmiarowa ilość wykrytych krawędzi na podłożu w danym obrazu w stosunku do krawędzi wykrytych na karoserii pojazdu.

Redukcję szumu na moim obrazie wykonałem w sposób dwuetapowy. W pierwszym kroku dokonałem konwersji obrazu w odcienie szarości. Implementacja tej funkcjonalności znajduje się w metodzie *ToGrayScale* klasy *FImage*. Polega ona zasadniczo na przypisaniu danemu pikselowi wartości wyliczonej za pomocą poniższej formuły:

$$\text{grayscaleMatrix}[i,j] = 0.299 * \text{_red}[i,j] + 0.578 * \text{_green}[i,j] + 0.114 * \text{_blue}[i,j]$$

gdzie *_red[i,j]*, *_green[i,j]*, *_blue[i,j]* to wartości koloru w konkretnych kanałach.

Następnie dokonałem rozmycia obrazu za pomocą filtru Gaussa. Funkcja *MakeGaussOperator* z klasy *GaussOperator* dla każdego piksela na obrazie liczy jego nową wartość korzystając z macierzy *_gaussianKernel* która tworzona jest podczas inicjalizacji obiektu klasy *GaussOperator*. Wartości poszczególnych elementów macierzy wyznaczane są za pomocą poniższej formuły

$$H[i,j] = \frac{1}{2 * \pi * \sigma^2} * e^{\frac{-(x^2+y^2)}{2*\sigma^2}}$$

Następnie dokonuję normalizacji macierzy tak aby suma wszystkich jej elementów była równa 1. Mając już utworzoną macierz filtru Gaussa wyznaczam nową wartość danego piksela poprzez pomnożenie wartości otaczających go pikseli przez odpowiednie elementy macierzy. Podczas implementacji funkcjonalności zmierzyłem się z problemem zachowania na krawędziach i problemem wychodzenia poza zakres pikseli w obrazie. Aby go rozwiązać wprowadziłem powiększanie przetwarzanego obrazu tak aby filtr Gaussa swoim działaniem objął wszystkie piksele w obrazie pierwotnym. Po dokonaniu operacji redukcji szumu zmniejszam obraz przywracając go do rozmiarów pierwotnych

Obraz powstały po redukcji szumu

Do demonstracji działania redukcji szumu wykorzystałem obraz poddając go konwersji w skalę szarości oraz rozmyciu za pomocą filtru Gaussa o szerokości macierzy równej 9 i wartości σ równej 3. Podczas konwersji obrazu w kolejnych etapach będę wykorzystywał macierz Gaussa o szerokości równej 5 ale dla wizualnego pokazania działaniu operacji rozmycia użyłem znacznie wyższej wartości przy której rozmycie jest widoczne gołym okiem natomiast powoduje to spadek w jakości i ilości wykrywanych krawędzi w późniejszych etapach działania algorytmu.



3.2 Szukanie natężenia gradientu obrazu

Po zredukowaniu szumów następnym etapem algorytmu jest wstępna detekcja krawędzi. Wykorzystywany przez mnie algorytm wykrywa krawędzie w czterech kierunkach: pion, poziom oraz przekątne. Do wstępnego wykrycia krawędzi wykorzystałem operator Sobela. Zwraca on wartość pierwszej pochodnej dla kierunku pionowego G_x i kierunku poziomego G_y . Następnie mając te dwie wartości wyznaczyłem gradient i kierunek krawędzi w danym punkcie wykorzystując poniższe wzory

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \frac{G_x}{G_y}$$

Realizacja szukania gradientu obrazu za pomocą filtru używającego operator Sobela znajduje się w klasie *SobelOperator*. Podczas tworzenia obiektu tej klasy przyjmowana jest macierz bajtów na której dokonywane będą obliczenia. Funkcja *MakeSobelOperator* zwraca macierz obiektów klasy *SobelPixel*. Obiekty tej klasy posiadają dwa parametry: natężenie gradientu oraz kąt nachylenia zgodnie z założeniami teoretycznymi. Są to wartości potrzebne w kolejnym etapie polegającym na usuwaniu niemaksymalnych pikseli.

Klasa *SobelOperator* zawiera dwie pomocnicze funkcje: *ValueGx* i *ValueGy* służące do obliczenia pierwszej pochodnej w kierunku pionowym i poziomym. Wartości pochodnych w danym punkcie wyznaczane są przy pomocy dwóch macierzy, każda o wymiarze 3x3 zatem tak jak w poprzednim kroku algorytmu spotykałem się z problemem zachowania na krawędziach. Aby go rozwiązać przed dokonaniem wstępnego wykrycia krawędzi poprzez szukanie natężenia obrazu ponownie go powiększyłem, tym razem o dokładnie jeden piksel zapewniając możliwość rozpatrywania wartości na krawędziach. W odróżnieniu od operacji redukcji szumu pomniejszenie obrazu do rozmiarów pierwotnych nastąpi dopiero po ostatnim etapie przetwarzania.

Obraz powstały po zastosowaniu filtru używającego operator Sobel



3.3 Usuwanie niemaksymalnych pikseli

Trzeci krok algorytmu Canny polega na usuwaniu niemaksymalnych pikseli w taki sposób, aby na obrazie znajdowały się jedynie krawędzie o szerokości jednego piksela. Funkcjonalność tą zapewnia metoda *MakeNonMaximumSuppression* znajdująca się w klasie *NonMaximumSuppression*. Operuje ona na macierzy obiektów *SobelPixel* stworzonej w poprzednim punkcie. Dla każdego elementu macierzy dokonuje ona odpowiednio wyboru rozpatrywanych sąsiadów w zależności od wartości kąta jaki posiada dany obiekt macierzy (rozpatrywanie następuje w czterech różnych kierunkach) a następnie porównania wartości natężenia gradientu rozpatrywanego piksela z wartościami gradientu wybranych pikseli. Jeżeli rozpatrywany piksel ma większą wartość niż oba porównywane to zostanie mu przypisana jego wartość (z ograniczeniem do 255 gdyż wartości zostaną wpisane do tablicy bajtów) lub zostanie mu przypisana wartość 0 jeżeli którykolwiek z rozpatrywanych sąsiadów będzie posiadać większą lub równą wartość gradientu natężenia.

Obraz powstały w wyniku usunięcia niemaksymalnych pikseli



3.4 Progowanie z histerezą

Ostatnim etapem algorytmu Canny jest progowanie z histerezą polegające na usunięciu nieistotnych krawędzi. Następuje to poprzez podziale pikseli w obrazie na trzy kategorie: silne, słabe i nieistotne. Następnie na pikselach słabych dokonywane są działania polegające na sprawdzeniu, czy sąsiaduje pośrednio lub bezpośrednio z pikselem silnym. Jeżeli tak to zostaje zamieniony na silny, jeżeli nie zostaje zamieniony na słaby. Po zakończeniu etapu sprawdzania i zamianiania rodzajów pikseli, tym oznaczonym jako słabe zostaje przypisana wartość 0 zatem stają się czarne, a tym oznaczonym jako silne zostaje przypisana wartość 1 zatem stają się białe.

Progowanie z histerezą realizowane jest w moim projekcie w dwóch klasach. W pierwszej klasie *DoubleThreshold* podczas tworzenia jej instancji przekazywane są dodatkowo dwa parametry określające progi słabego i silnego piksela. Funkcją *CreateDoubleThresholdMatrix* na ich podstawie każdemu pikselowi przypisuje dane oznaczenie. Następnie w klasie *HysteresisEdgeTracking* metoda *EdgeTracking* dokonuje śledzenia i zamiany słabych pikseli zgodnie z omówioną zasadą. Po dokonaniu tych działań otrzymywana jest macierz bajtów która jest naszą macierzą wyjściową na podstawie, której bezpośrednio powstanie obraz końcowy

Obraz końcowy po procesie wykrywania krawędzi



4 Opis uruchomienia programu

Stworzony przeze mnie program wywoływane jest poprzez komendę za pomocą polecenia *dotnet run* i przyjmuje 6 argumentów. Dwa pierwsze z nich są argumentami niezbędnymi, są to odpowiednio ścieżka do obrazu przetwarzanego i ścieżka do obrazu wyjściowego który ma zostać stworzony i zapisany pod zadaną nazwą. Użytkownik dodatkowo może podać cztery opcjonalne parametry:

- wartość parametru sigma, -s, -sigma (liczba zmiennoprzecinkowa dodatnia)
- szerokość macierzy Gaussa, -g, -gauss (liczba dodatnia nieparzysta)
- wartość granicy dolnej, -l, -lower (liczba całkowita z zakresu 1-255)
- wartość granicy górnej, -u, -upper (liczba całkowita z zakresu 1-255)

Jeżeli podane argumenty nie zostaną podane przez użytkownika wykorzystane zostaną wartości domyślne powyższych parametrów (-s 3.0, -g 5, -l 60, -u 100).

Przykład wywołania programu

```
dotnet run projekt.jpeg out.png -s 2 -g 7 -l 100 -u 200
```

5 Przykłady działania programu

Wykrywanie krawędzi za pomocą stworzonego przeze mnie programu przetestowałem na obrazach o różnej charakterystyce, różniących się ilością detali, wyrazistością krawędzi czy poziomem kontrastu. Dodatkowo sprawdziłem wpływ poszczególnych parametrów na efekty działania. Poniżej zamieszczam kilka przykładów pokazujących efekty wykrywania krawędzi

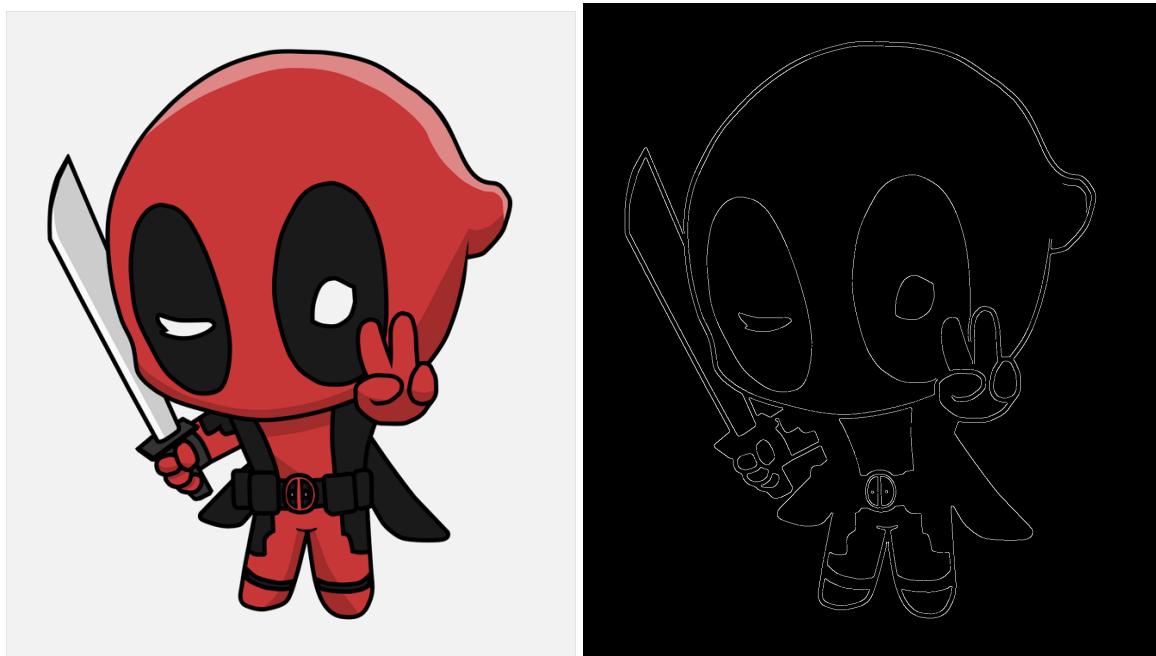
5.1 Obraz o dużej ilości szczegółów

Parametry wywołania przetwarzania obrazu: -s 3 -g 7 -l 80 -u 100



5.2 Obraz animowany o dobrze widocznych krawędziach

Parametry wywołania przetwarzania obrazu: -s 2 -g 5 -l 100 -u 160



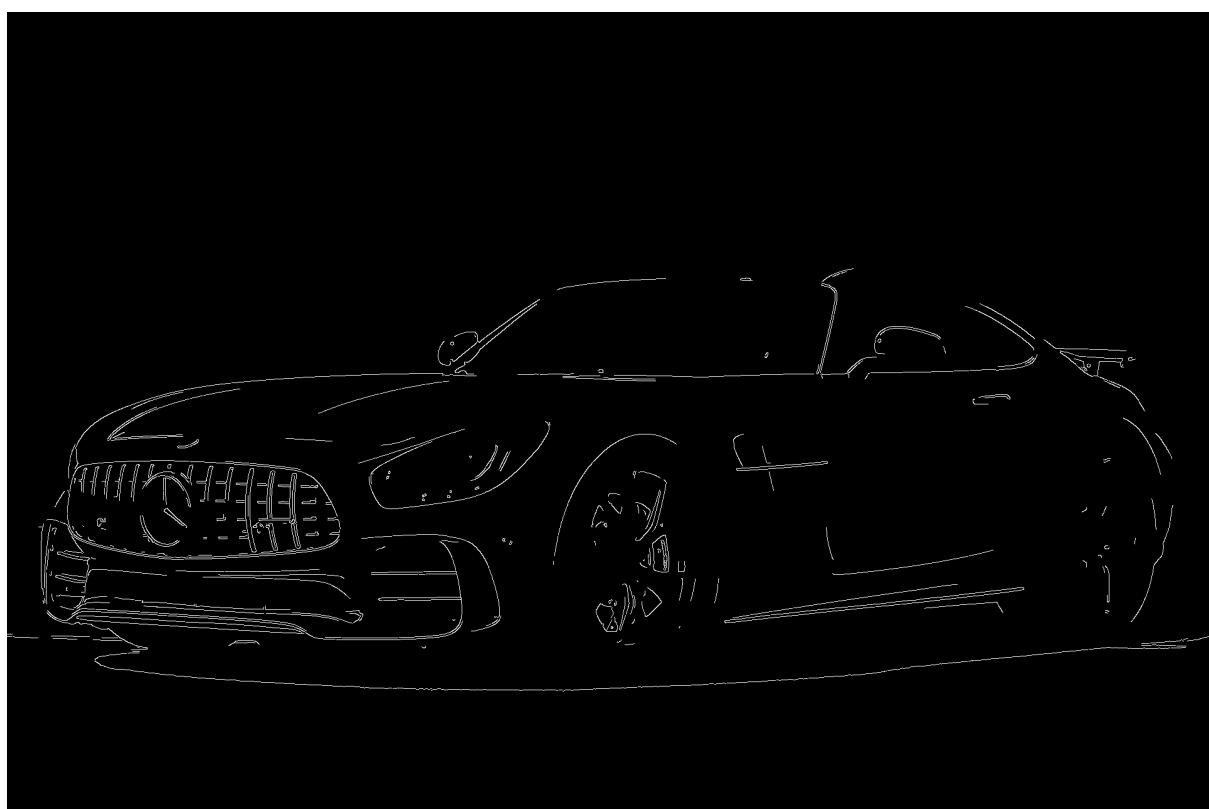
5.3 Porównanie efektów działania programu dla różnych wartości parametrów początkowych



Bardzo małe rozmycie i duża ilość detali: -s 1 -g 1 -l 30 -u 60



Bardzo niewielkie rozmycie i zaznaczenie wyłącznie najwidoczniejszych krawędzi: -s 1 -g 1 -l 30 -u 60



Duże rozmycie i bardzo duża ilość detali: -s 5 -g 7 -l 1 -u 20



6 Podsumowanie

Praca nad algorytmem pomimo kilku trudności zakończyła się powodzeniem. Stworzony przeze mnie program jest w stanie na podstawie dostarczonego obrazu wygenerować obraz z wyraźnie zaznaczonymi krawędziami. Jakość rezultatów i ilość wykrytych krawędzi w dużym stopniu zależy od rodzaju obrazu oraz parametrów początkowych programu. Użytkownik za pomocą zmiany poziomu rozmycia czy ustalania dolnej i górnej granicy podczas progowania jest w stanie dostosować ilość i jakość wykrytych krawędzi do jego potrzeb. Program działa dobrze dla obrazów, na których krawędzie są wyraziste. Przetwarzanie obrazów o dużym natężeniu i wysokiej ilości niewielkich krawędzi również działa poprawnie. Problemem może być jedynie jakość odbioru obrazu przez użytkownika, gdyż takie obrazy tracą przejrzystość przez nagromadzenie dużej ilości krawędzi blisko siebie co spotęgowane jest oznaczeniem ich wyłączenie w sposób biało czarny. Daje to wrażenie zgęstnienia i przesytu.

7 Źródła

https://en.wikipedia.org/wiki/Canny_edge_detector

Wszystkie obrazy z jakich korzystałem pochodzą ze źródeł wolnych i nie są obcięte prawami autorskimi

