Architektura komputerów

Laboratorium 5 w terminie 24 maja 2021

Jakub Superczyński 241381

7 czerwca 2021

1 Cel laboratorium

Celem laboratorium było napisanie w języku assemblera w połączeniu z C w architekturze 32-bitowej na platformę Linux programu do filtrowania obrazów. Należało zaimplementować i przetestować 3 różne filtry. Dla ułatwienia zaimplementowałem 3 filtry o takim samym rozmiarze maski i testowałem program tylko na bitmapach kodujących jeden piksel na 24 i 32 bitach

2 Opis implementacji

2.1 Wczytywanie i zapisywanie do pliku

Działanie na plikach bmp realizowane jest z poziomu C analogicznie do sposobu z poprzedniego laboratorium. Jedyną różnicą jest zapis, ponieważ tym razem nie nadpisuję pliku wejściowego, ale tworzę inny plik. Aby zachować poprawny format pliku, należało przekopiować również wszystkie metadane, a do tego musiałem obliczyć rozmiar dodatkowych nagłówków. Rozmiar części przed tablicą pikseli to po prostu offset pikseli, więc jest mi już znana. Większy problem stanowi część po tablicy pikseli. Aby poznać jej rozmiar, musiałem wczytać rozmiar całego pliku w bajtach a następnie odjąć od niego offset pikseli oraz ich rozmiar w bajtach. Mając już te informacje, wczytanie pliku i wywołanie funkcji assemblerowej z odpowiednimi argumentami jest bardzo proste.

2.2 Piksele krańcowe

Ważną częścią filtrowania jest omijanie pikseli krańcowych. W przeciwnym razie program próbowałby pobrać zmienne które leżą poza dostępną pamięcią. Dlatego na samym początku zamieniam cały pierwszy rząd pikseli na białe piksele. Jest to o tyle proste, że do funkcji przesyłam uprzednio obliczoną liczbę bajtów na rząd *bpr*. Zamieniam więc bajt po bajcie w pętli wykonującej się *bpr* razy, z każdym krokiem zwiększając adres wejściowego i wyjściowego piksela. Po zamianie pierwszego rzędu rozpoczynam główną pętlę, która będzie wykonywała się o 2 mniej niż wynosi wysokość obrazka (aby zamienić wszystkie rzędy oprócz

ostatniego oraz pierwszego, który jest już zamieniony). Każda iteracja tej pętli to kolejny rząd, dlatego na początku należy ominąć pierwszy piksel. Robię to w pętli wykonującej się tyle razy ile bajtów koduje jeden piksel (podobnie jak postępowałem z rzędem, do funkcji przekazuję ilość bajtów na piksel *bpr*, jest to liczba iteracji tej pętli). W pętli tej zamieniam krańcowy piksel na kolor biały. Następnie rozpoczynam kolejną pętlę, wykonującą się o dwa mniej niż szerokość obrazka (znowu - dla wszystkich kolumn oprócz pierwszej i ostatniej), w której wykonuję algorytm filtrowania. Po zakończeniu tej pętli zamieniam ostatni piksel. Gdy przedostatni rząd zostanie przefiltrowany, należy jeszcze zamienić ostatni zamienić na kolor biały.

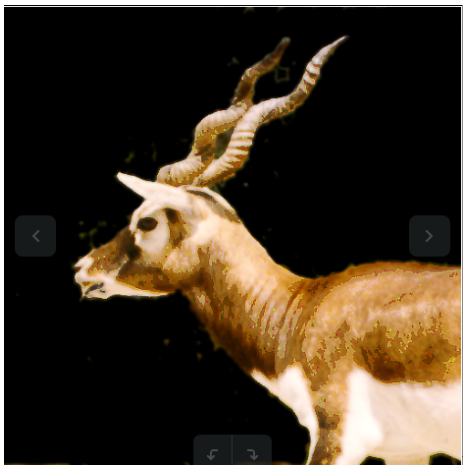
2.3 Filtrowanie

Algorytm filtrowania składa się tak naprawdę z prostych mnożeń, dodawań i dzielenia. Prawdziwą trudnością jest obliczenie adresów pikseli które biorą udział w danym filtrowaniu. W tym celu funkcja przyjmuje jako argument ilość bajtów na piksel *bpp* oraz liczbę bajtów na rząd *bpr*. Znając te wartości oraz posiadając adres obecnie filtrowanego piksela, dość łatwo mogę policzyć wszystkie interesujące mnie piksele. Np. aby otrzymać adres bajta w lewym górnym rogu maski, muszę odjąć od aktualnego adresu *bpr* oraz *bpp*. Z każdym z 9 bajtów postępuję analogicznie, aby je wydobyć. Następnie mnożę go przez wartość z maski i dodaję do siebie. Po tym sumuję wagi maski i jeśli ta suma jest różna od zera, dzielę sumę ważoną wartości piksela przez sumę wag maski. Tak otrzymaną wartość mogę zapisać do odpowiedniego miejsca pikseli wyjściowych

3 Przykład działania

Obrazek bez filtru:

Filtr uśredniający:



Filtr usuwający średnią:

Filtr konturowy:

4 Podsumowanie

Program, choć podczas debugowania i sprawdzania wartości pliku, wydaje się działać poprawnie (tzn. zgodnie z algorytmem), wizualnie nie daje dobrych efektów. Wydaje się że jednak efekt działania jest dość odległy od idealnego filtru. Nie jestem pewien co jest tego powodem: być może specyficzne rysunki wejściowe, być może błędne zrozumienie algorytmu filtrowania.
Okazało się, że sam algorytm filtrowania jest dość prosty. Problem stanowiło ominięcie krańcowych pikseli oraz znalezienie sposobu na obliczenie adresów bajtów podstawianych do maski. Ostatecznie zdecydowałem się na sposoby dość trywialne, choć przez to może nie do końca efektywne, jednak łatwe do sprawdzenia.